# Introduction to Automata and Process Theory

E.P. de Vink

Department of Mathematics and Computer Science
Eindhoven University of Technology

February 10, 2015

# Contents

# Preface

This reader is based on the following material:

- J.E. Hopcroft, R. Motwani , J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, (2nd ed.), Addison-Wesley 2001.
- H. Lewis, C.H. Papadimitriou, *Elements of the Theory of Computation*, (2nd ed.), Prentice Hall 1998
- M. Sipser, *Introduction to the Theory of Computation*, (2nd ed.), PWS 1997
- J.C.M. Baeten, *Models of Computation: Automata, Formal Languages and Communicating Processes*, manuscript, 2013

# Chapter 1

# Preliminaries

## Strings

A non-empty finite set $\Sigma$ is called an alphabet, its elements are called letters or symbols. Typically we use $a$ and $b$ to range over $\Sigma$.

**Definition 1.1.** Let $\Sigma$ be an alphabet. The set $\Sigma^*$ of strings or finite words over $\Sigma$ is defined as follows:

- the empty string $\varepsilon \in \Sigma^*$;
- if $a \in \Sigma$ and $w \in \Sigma^*$ then $aw \in \Sigma^*$.

A string $w \in \Sigma^*$ can be uniquely written as $w = a_1 \cdots a_n$ for some $n \geqslant 0$. Hence,

$$\Sigma^* = \{\, a_1 a_2 \cdots a_n \mid n \geqslant 0,\ \forall i, 1 \leqslant i \leqslant n \colon a_i \in \Sigma \,\}$$

**Definition 1.2.** Given an alphabet $\Sigma$, the length $|w|$ for a string $w \in \Sigma^*$ is given by: (i) $|\varepsilon| = 0$, (ii) $|aw| = |w| + 1$.

Thus, a string $w = a_1 \cdots a_n$ is said to have length $n$. For $n \geqslant 0$ we put $\Sigma^n = \{\, w \in \Sigma^* \mid |w| = n \,\}$, i.e. $\Sigma^n$ consists of the strings over $\Sigma$ of length $n$. Although $\Sigma$ is a finite set, and each set $\Sigma^n$ is finite, the set $\Sigma^*$ is infinite. In fact, $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$.

**Example 1.3.** Let $\Sigma = \{a, b\}$ be a two-letter alphabet with symbols $a$ and $b$. Then

$$\begin{array}{lcl}
\Sigma^0 & = & \{\varepsilon\} \\
\Sigma^1 & = & \{a, b\} \\
\Sigma^2 & = & \{aa, ab, ba, bb\} \\
\Sigma^3 & = & \{aaa, aab, aba, abb, baa, bab, bba, bbb\} \\
 & \cdots & \\
\Sigma^* & = & = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \ldots\}
\end{array}$$

Below, we mainly apply an induction principle based on the length function for proving properties of strings. The base case is $|w| = 0$, i.e. $w = \varepsilon$. The induction step is to prove the property for strings of length $n+1$ assuming the property holds for strings of length $n$ or less. If $|w| = n+1$ for $n \geqslant 0$, we can write $w = av$ for a symbol $a$ and a string $v$ while $|v| = n$.

**Definition 1.4.** Let $\Sigma$ be an alphabet. The concatenation $wv \in \Sigma^*$ of strings $w, v \in \Sigma^*$ is given by: (i) $\varepsilon v = v$, and (ii) $(aw)v = a(wv)$.

Thus, for two strings $w = a_1 \cdots a_n$ and $v = b_1 \cdots b_m$ in $\Sigma^*$ for $n, m \geqslant 0$, where $a_1, \ldots, a_n, b_1, \ldots, b_m \in \Sigma$, the concatenation $wv \in \Sigma^*$ is given by $wv = c_1 \cdots c_\ell$ where $\ell = n + m$ and

$$\begin{cases} c_i = a_i & \text{for } 1 \leqslant i \leqslant n \\ c_{n+j} = b_j & \text{for } 1 \leqslant j \leqslant m \end{cases}$$

If $w = aw'$, i.e. $w$ has a first symbol $a$ and remainder $w'$, we can also see the string $w$ as the concatenation of the one-symbol string $a$ and the string $w'$. As this all amounts to the same, no distinguishing notation is used for concatenation of strings.

From the definition it follows that $w\varepsilon = w$ for all $w \in \Sigma^*$ by induction on $|w|$.

Basis, $|w| = 0$: We have $w = \varepsilon$. Thus, by clause (i) of Definition 1.4, $w\varepsilon = \varepsilon\varepsilon = \varepsilon = w$. Induction step, $|w| = n+1$: We have $w = aw'$ for a symbol $a \in \Sigma$ and some string $w' \in \Sigma^*$ with $|w'| = n$. We have, by clause (ii) of Definition 1.4, $w\varepsilon = (aw')\varepsilon = a(w'\varepsilon) = aw' = w$ since $w'\varepsilon = w'$ by induction hypothesis.

It follows that $\varepsilon$ is a unit element for $\Sigma^*$ with respect to concatenation, i.e. $\varepsilon v = v$ and $w\varepsilon = w$ for all strings $v, w$. A similar role is played by 0 for the integers with respect to addition. There we have $0 + y = y$ and $x + 0 = x$ for all integers $x, y$. To draw the comparison slightly further, for addition of integers $x$, $y$ and $z$, the order in which addition takes place for $x + y + z$ is not relevant to the end result since $(x + y) + z = x + (y + x)$. Similarly, concatenation of strings is associative, i.e. the resulting string is the same for any order of application. Thus, for some alphabet $\Sigma$, $(wv)u = w(vu)$ for all strings $w, v, u \in \Sigma^*$. Note, addition of integer is commutative, i.e. $x + y = y + x$ for integers $x, y$, but this does not hold in general for strings over an alphabet of a least two symbols.

A string $v$ is called a *prefix* of a string $w$ if $vu = w$ for some string $u$, notation $v \preccurlyeq w$. Thus extending $v$ with $u$, i.e. taking the concatenation of $v$ and $u$, yields $w$. The string $u$ is uniquely determined by the strings $v$ and $w$. In the situation that $v \preccurlyeq w$ we occasionally write $u = w/v$. Thus, if $v \preccurlyeq w$ then we have $v\,(w/v) = w$.

If $vu = w$ for some string $v$, then the string $u$ is called a *suffix* of the string $w$, notation $w \succcurlyeq u$. If $w = xyz$ for strings $x, z$, then the string $y$ is called a *substring* of the string $w$. Note, in view of the associativity of concatenation no parentheses are needed in writing $w = xyz$. It is always the case that $\varepsilon$ and $w$ are prefixes, suffixes and substrings of $w$.

**Definition 1.5.** Let $\Sigma$ be an alphabet and $c \in \Sigma$. The count $\#_c(w)$ of a symbol $c \in \Sigma$ in a string $w \in \Sigma^*$ is given by

$$\#_c(\varepsilon) = 0 \qquad \#_c(cw) = \#_c(w) + 1 \qquad \#_c(aw) = \#_c(w) \text{ if } a \neq c$$

So far we considered operations on strings. Next we consider operations on sets of strings.

**Definition 1.6.** Let $\Sigma$ be an alphabet. A subset $L \subseteq \Sigma^*$ is called a language over $\Sigma$.

**Example 1.7.**

- $\emptyset$, $\{\varepsilon\}$ and $\Sigma^*$ are languages, for every alphabet $\Sigma$.
- For the alphabet $\Sigma = \{a, b\}$, the set $L_1 = \{\, a^n b^n \mid n \geqslant 0 \,\}$ is an example language, as is $L_2 = \{\, w \in \Sigma^* \mid \#_a(w) = \#_b(w) \,\}$. It holds that $\emptyset \subseteq L_1 \subseteq L_2 \subseteq \Sigma^*$.

**Definition 1.8.** Let $L_1, L_2, L \subseteq \Sigma^*$ be three languages over an alphabet $\Sigma$. The concatenation $L_1 \cdot L_2$ of $L_1$ and $L_2$ is given by

$$L_1 \cdot L_2 \;=\; \{\, w_1 w_2 \mid w_1 \in L_1,\; w_2 \in L_2 \,\}$$

The Kleene-closure $L^*$ of $L$ is given by

$$L^* \;=\; \{\, w_1 \dots w_n \mid n \geqslant 0,\; w_1, \dots, w_n \in L \,\}$$

For an alphabet $\Sigma$ we have that $\Sigma \cdot \Sigma = \Sigma^2$. Moreover, the Kleene-closure $\Sigma^*$ of $\Sigma$ coincides with the set $\Sigma^*$ of strings over $\Sigma$ given in Definition 1.1 coincide.

# Chapter 2

# Finite Automata and Regular Languages

In this chapter we introduce the notion of a deterministic finite automaton, of a non-deterministic finite automaton with silent steps and of a regular expression. We will show that the class of associated languages, the class of regular languages, for all these three concepts is the same. We study closure properties of the class of regular languages and provide a means to prove that a language is not regular.

## 2.1 Deterministic finite automata

We start off with the simplest yet most rigid concept of the three main notions mentioned.

**Definition 2.1** (Deterministic finite automaton)**.** A deterministic finite automaton (DFA) is a tuple $D = (Q, \Sigma, \delta, q_0, F)$ with $Q$ a finite set of states, $\Sigma$ a finite alphabet, $\delta : Q \times \Sigma \to Q$ the transition *function*, $q_0 \in Q$ the initial state, and $F \subseteq Q$ the set of final states.

We sometimes write $q \xrightarrow{a}_D q'$ instead of $\delta(q, a) = q'$, and call it a transition of $D$ from $q$ to $q'$ labeled $a$. We then may write $q \xrightarrow{a} q'$ if the automaton $D$ is clear from the context. Intuitively, when automaton $D$ is in state $q$ and reads the symbol $a$ from the input, it moves to state $q'$.

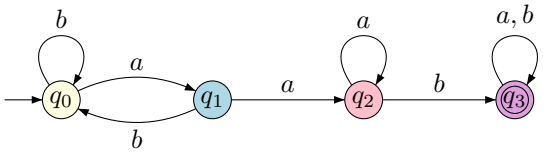



Figure 2.1: Finite automaton of Example 2.2

**Example 2.2.** Figure 2.1 gives a visual representation of an example deterministic finite automaton. The set of states is $\{ q_0, q_1, q_2, q_3 \}$ with $q_0$ the initial state as indicated by the small incoming arrow. The alphabet $\Sigma$ consists of the symbols $a$ and $b$. The $\delta$-function is indicated by the arrows between states. E.g., for $q_0$ there is an arrow labeled $a$ to $q_1$, thus $\delta(q_0, a) = q_1$. There is also an arrow labeled $b$ from $q_0$ to itself, so $\delta(q_0, b) = q_0$. The self-loop of $q_3$ labeled $a, b$ represents two transitions, one for $a$ and one for $b$. Thus $\delta(q_3, a) = q_3$ and $\delta(q_3, b) = q_3$. There is one final state, viz. $q_3$, as indicated by the double boundary of the state. Thus, the set of final states is $\{q_3\}$.

A configuration of a finite automaton $D = (Q, \Sigma, \delta, q_0, F)$ is a pair of a state $q \in Q$ and a string $w \in \Sigma^*$, notation $(q, w)$. The configuration $(q, w)$ indicates that $D$ is in state $q$ with the word $w$ on input. We write $(q, w) \vdash_D (q', w')$ if automaton $D$ in state $q$ moves to state $q'$ when reading the first symbol of $w$. More formally, the relation $\vdash_D \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ is defined by

$$(q, w) \vdash_D (q', w') \quad \text{iff} \quad w = aw' \text{ and } \delta(q, a) = q' \text{ for some } a \in \Sigma$$

We say that $(q, w)$ yields $(q', w')$ with respect to $D$.

Note $\vdash_D \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ is a relation on $Q \times \Sigma^*$. We denote by $\vdash_D^*$ the reflexive and transitive closure of $\vdash_D$. Thus

$$\begin{aligned}
&(q, w) \vdash_D^* (q', w') \quad \text{iff} \\
&\quad \exists n \geqslant 0\, \exists w_0, \ldots, w_n \in \Sigma^*\, \exists q_0, \ldots, q_n \in Q\colon \\
&\qquad (q, w) = (q_0, w_0),\ (q_{i-1}, w_{i-1}) \vdash_D (q_i, w_i),\ \text{for } 1 \leqslant i \leqslant n, \\
&\qquad\quad \text{and } (q_n, w_n) = (q', w')
\end{aligned}$$

For a DFA we have the following two properties.

**Lemma 2.3.** Let $D$ be a DFA.

(a) For all states $q, q', q''$ and words $w, w'$ it holds that
if $(q, w) \vdash_D^* (q', w')$ and $(q, w) \vdash_D^* (q'', w')$ then $q' = q''$.

(b) For states $q, q'$ and all words $w, w', v$ it holds that
$(q, w) \vdash_D^* (q', w')$ iff $(q, wv) \vdash_D^* (q', w'v)$.

*Proof.* Left as an exercise. □

The first item expresses the determinacy of the DFA $D$: the state and input determine the resulting state. The second item expresses that a computation starting from a particular state of the DFA is for the rest only influenced by the input.

**Definition 2.4.** Let $D = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton. The language $\mathcal{L}(D) \subseteq \Sigma^*$ accepted by $D$ is defined by

$$\mathcal{L}(D) = \{ w \in \Sigma^* \mid \exists q \in F\colon (q_0, w) \vdash_D^* (q, \varepsilon) \}$$

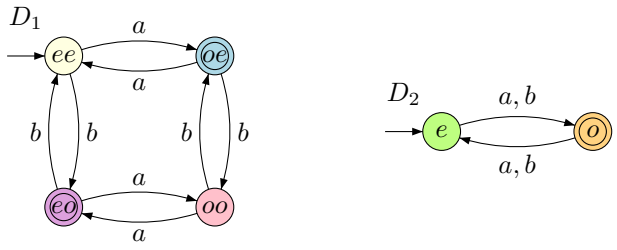



Figure 2.2: DFAs of Example 2.6

**Example 2.5.** We claim that for the automaton $D$ of Example 2.2, depicted in Figure 2.1, we have

$$\mathcal{L}(D) \;=\; \{\, w \in \{a,b\}^* \mid w \text{ has a substring } aab \,\}$$

To see this we make use of so-called path sets. We say that $L$ is the path set reaching $q$ with respect to $D$ iff $L$ contains exactly those words $w$ that bring, when starting from the initial state, the automaton to state $q$. Notation $pathset_D(q)$. Thus

$$pathset_D(q) = \{\, w \in \Sigma^* \mid (q_0, w) \;\vdash_D^*\; (q, \varepsilon) \,\}$$

For this example we have

| state | path set |
|---|---|
| $q_0$ | no substring $aab$, not ending in $a$ |
| $q_1$ | no substring $aab$, ending in $a$, not in $aa$ |
| $q_2$ | no substring $aab$, ending in $aa$ |
| $q_3$ | substring $aab$ |

Since $q_3$ is the only final state of $D$ it follows that a string $w \in \{a,b\}^*$ is accepted by $D$ iff $w$ has a substring $aab$, as was to be shown.

**Example 2.6.** Consider the deterministic finite automaton $D_1$ given by the left part of Figure 2.2. The language $\mathcal{L}(D_1)$ of $D_1$ is the set

$$\mathcal{L}(D_1) = \{\, w \in \{a,b\}^* \mid \#_a(w) \text{ odd, or } \#_b(w) \text{ odd, but not both} \,\}$$

For $D_1$ we have the following path sets

| state | path set |
|---|---|
| $q_{ee}$ | $\#_a(w)$ even, $\#_b(w)$ even |
| $q_{oe}$ | $\#_a(w)$ odd, $\#_b(w)$ even |
| $q_{eo}$ | $\#_a(w)$ even, $\#_b(w)$ odd |
| $q_{oo}$ | $\#_a(w)$ odd, $\#_b(w)$ odd |

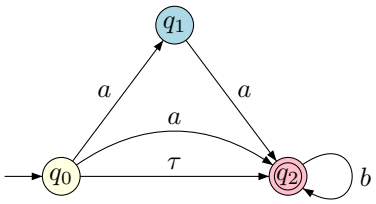



Figure 2.3: Example NFA

Only $q_{oe}$ and $q_{eo}$ are final states, thus $w$ is accepted iff $\#_a(w)$ odd and $\#_b(w)$ even, or $\#_a(w)$ even and $\#_b(w)$ odd.

Now consider the deterministic finite automaton $D_2$ given by the right part of Figure 2.2. The state $q_e$ is the initial state, the state $q_o$ the only final state. We claim that the language $\mathcal{L}(D_2)$ of strings that are accepted by $D_2$ is the set of strings of odd length, i.e.

$$\mathcal{L}(D_2) = \{\, w \in \{a, b\}^* \mid |w| \text{ odd} \,\}$$

as can be seen by computing the path sets for $q_e$ and $q_o$:

| state | path set |
|---|---|
| $q_e$ | $\lvert w\rvert$ even |
| $q_o$ | $\lvert w\rvert$ odd |

Since $q_o$ is the only final state of $D_2$ the claim follows.

Note that $D_1$ and $D_2$ accept the same language, i.e. $\mathcal{L}(D_1) = \mathcal{L}(D_2)$.

## 2.2 Finite automata

A DFA has for each state exactly one outgoing transition for each symbol. A non-deterministic finite automaton with silent steps can have any number of outgoing transitions for a given symbol, including no transitions. Additionally, transitions are allowed that do not need input. These are the so-called silent steps.

**Definition 2.7** (Non-deterministic finite automaton with silent steps)**.** A non-deterministic finite automaton with silent steps (NFA), or just a finite automaton, is a quintuple $N = (Q, \Sigma, \rightarrow_N, q_0, F)$ with $Q$ a finite set of states, $\Sigma$ a finite alphabet, $\rightarrow_N \subseteq Q \times (\Sigma \cup \{\tau\}) \times Q$ the transition *relation*, $q_0 \in Q$ the initial state, and $F \subseteq Q$ the set of final states.

Instead of a function $\delta : Q \times \Sigma \to Q$ as we have for a DFA, we consider for an NFA a relation $\rightarrow_N$. This relation may include triples $(q, a, q')$ for states $q, q' \in Q$ and a symbol $a \in \Sigma$, but also triples $(q, \tau, q')$ for states $q, q' \in Q$ and the special symbol $\tau$. The symbol $\tau$ denotes a so-called silent step. We write $\Sigma_\tau$ for $\Sigma \cup \{\tau\}$. It is assumed

that $\tau \notin \Sigma$. We also write $q \xrightarrow{\alpha}_N q'$ if $(q, \alpha, q') \in \rightarrow_N$, thus with $\alpha \in \Sigma_\tau$. $q \xrightarrow{\tau}_N q'$ we say that there is a silent step or $\tau$-transition in $N$ from $q$ to $q'$. We often omit the subcript $N$ when clear from the context.

The concepts of an NFA and of a DFA are very similar, but differ in three aspects:

- in an NFA states can have $\tau$-transitions;
- in an NFA states can have multiple transitions for the same symbol;
- in an NFA states can have no transitions for a symbol.

Figure 2.3 gives a visual representation of a non-deterministic finite automaton. State $q_0$ has a $\tau$-transition to state $q_2$, has two transitions on symbol $a$ (one going to $q_1$, and one going to $q_2$) and has no transition on symbol $b$.

The transition relation $\rightarrow_N$ contains the triples

$$(q_0, \tau, q_2) \qquad (q_0, a, q_1) \qquad (q_0, a, q_2) \qquad (q_1, a, q_2) \qquad (q_2, b, q_2)$$

Still we can interpret $\rightarrow_N$ as function, say the function $\hat{\delta}$, but now of type $Q \times \Sigma_\tau \rightarrow \mathcal{P}(Q)$, with $\mathcal{P}(Q)$ the powerset of $Q$, the collection of all the subsets of $Q$. Here we have

$$\begin{array}{lll} \hat{\delta}(q_0, a) = \{q_1, q_2\} & \hat{\delta}(q_1, a) = \{q_2\} & \hat{\delta}(q_2, a) = \emptyset \\ \hat{\delta}(q_0, b) = \{q_2\} & \hat{\delta}(q_1, b) = \emptyset & \hat{\delta}(q_2, b) = \{q_2\} \\ \hat{\delta}(q_0, \tau) = \{q_2\} & \hat{\delta}(q_1, \tau) = \emptyset & \hat{\delta}(q_2, \tau) = \emptyset \end{array}$$

Note $\hat{\delta}(q_0, b) = \{q_2\}$. The idea is to combine the $\tau$-transition $q_0 \xrightarrow{\tau} q_2$ with the $b$-transition $q_2 \xrightarrow{b} q_2$. We come back to this idea for the general situation at a later stage.

As for deterministic finite automata we have the notion of a configuration $(q, w)$ for $q \in Q$ and $w \in \Sigma^*$. Also here we have the yield relation among configurations. We put

$$(q, w) \vdash_N (q', w') \quad \text{iff} \quad \exists a \in \Sigma \colon q \xrightarrow{a}_N q' \wedge w = aw' \text{ or } q \xrightarrow{\tau}_N q' \wedge w = w'$$

Note, if $q \xrightarrow{\tau}_N q'$ then we have $(q, w) \vdash_N (q', w)$ without a change of the remaining input $w$.

With respect to finite automaton of Figure 2.3 we have, e.g., $(q_0, abb) \vdash_N^* (q_2, \varepsilon)$ and $(q_0, abb) \vdash_N^* (q_1, bb)$. Also $(q_0, abb) \vdash_N^* (q_2, bb)$, $(q_2, bb) \vdash_N^* (q_2, b)$, and $(q_2, b) \vdash_N^* (q_2, \varepsilon)$. In configuration $(q_1, bb)$ the automaton $N$ cannot process the (first) input $b$; the automaton $N$ is stuck in that configuration. In contrast $(q_0, bb) \vdash_N^* (q_2, \varepsilon)$, since $(q_0, bb) \vdash_N (q_2, bb)$ via the silent step $q_0 \xrightarrow{\tau}_N q_2$, and $(q_2, bb) \vdash_N (q_2, b)$, $(q_2, b) \vdash_N (q_2, \varepsilon)$.

**Lemma 2.8.** For an NFA $N$ it holds that

$$(q, w) \vdash_N^* (q', w') \quad \text{iff} \quad (q, wv) \vdash_N^* (q', w'v)$$

for all words $w, w', v$ and states $q, q'$.

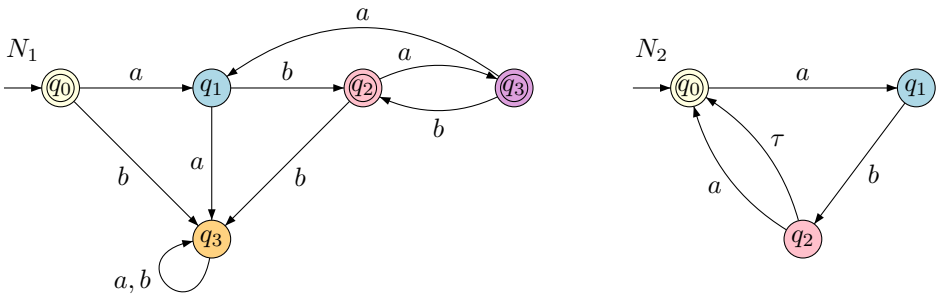



Figure 2.4: Finite automata for Example 2.10

*Proof.* Left as an exercise. □

The lemma of NFAs corresponds to Lemma 2.3, part (b) for DFAs. In the non-deterministic setting of NFAs there is no counterpart of Lemma 2.3, part (a). E.g., for the NFA of Figure 2.3 we have $(q_0, abb) \vdash_N^* (q_1, bb)$ and $(q_0, abb) \vdash_N^* (q_2, bb)$ while $q_1 \neq q_2$.

**Definition 2.9** (Language of an NFA)**.** Let $N = (Q, \Sigma, \rightarrow_N, q_0, F)$ be a finite automaton. The language $\mathcal{L}(N)$ accepted by $N$ is defined by

$$\mathcal{L}(N) = \{\, w \in \Sigma^* \mid \exists q \in F \colon (q_0, w) \vdash_N^* (q, \varepsilon) \,\}$$

A language accepted by a finite automaton is called a regular language.

Note the similarity of Definition 2.9 for the language of an NFA and the corresponding definition, Definition 2.4, for a DFA. Also note that $\mathcal{L}(N)$ for an NFA $N$ is considered a language over the alphabet $\Sigma$, rather than the alphabet $\Sigma_\tau$. Thus $\tau$-transitions do not contribute (directly) to the word that is accepted.

Two NFAs $N_1$ and $N_2$ are called language equivalent if they accept the same language, i.e. $\mathcal{L}(N_1) = \mathcal{L}(N_2)$. Likewise for two DFAs. We also call an NFA and a DFA language equivalent if they accept the same language.

**Example 2.10.** The two finite automata depicted in Figure 2.4 accept the same language. This is the language

$$\{ab, aba\}^* = \{\, w_1 \cdots w_n \mid n \geqslant 0, \forall i, 1 \leqslant i \leqslant n \colon w_i = ab \vee q_i = aba \,\}$$

However, the right automaton $N_2$ allows a $\tau$-transition. The left automaton $N_1$ can be seen as both as a DFA and as an NFA, the right automaton can be seen as an NFA only.

**Example 2.11.** Figure 2.5 provides an NFA that accepts decimal numbers. The three transitions from $q_0$ to $q_1$, with labels $+$, $-$ and $\tau$, express that the sign is optional; it can be a $+$-sign, a $-$-sign, or it can be omitted. In state $q_1$, on reading a digit from input, the automaton can move to state $q_4$ or stay in state $q_1$. In state $q_4$ the $\tau$-transition to

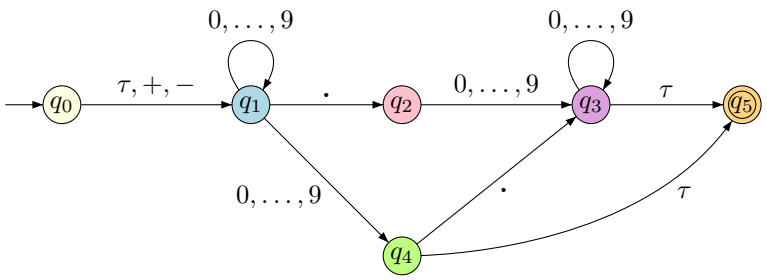



Figure 2.5: An NFA accepting decimal numbers

state $q_5$ can always be taken, leading to acceptance. Clearly, this latter transition is redundant, we could have left it out and could have made state $q_4$ final, instead. For the $\tau$-transition leaving state $q_0$ it may not be obvious if and how it can be eliminated. However, we shall prove that if a language is accepted by an NFA, then there is a DFA accepting it too.

**Theorem 2.12.** If a language $L \subseteq \Sigma^*$ is accepted by a DFA, then $L$ is also accepted by an NFA.

*Proof.* Suppose $L = \mathcal{L}(D)$ with $D = (Q, \Sigma, \delta, q_0, F)$ a deterministic finite automaton. Put $N = (Q, \Sigma, \to_N, q_0, F)$ with

$$q \xrightarrow{a}_N q' \quad \text{iff} \quad \delta(q, a) = q'$$

for $q, q' \in Q$, $a \in \Sigma$. Then it holds that $(q, w) \vdash_D (q', w')$ iff $(q, w) \vdash_N (q', w')$. Therefore we have, by Definition 2.4 and Definition 2.9, respectively,

$$\begin{aligned} & w \in \mathcal{L}(D) \\ \iff & \exists q \in F \colon (q_0, w) \vdash_D^* (q, \varepsilon) \\ \iff & \exists q \in F \colon (q_0, w) \vdash_N^* (q, \varepsilon) \\ \iff & w \in \mathcal{L}(N) \end{aligned}$$

Thus $\mathcal{L}(D) = \mathcal{L}(N)$ and $L$ is accepted by the NFA $N$. □

As is to be expected, the NFA $N$ constructed in the proof above does not involve silent steps.

The reverse of Theorem 2.12 holds as well.

**Theorem 2.13.** If a language $L \subseteq \Sigma^*$ is accepted by an NFA, then $L$ is also accepted by a DFA.

*Proof.* Suppose $L = \mathcal{L}(N)$ with $N = (\, Q_N,\, \Sigma,\, \to_N,\, q_N^0,\, F_N\,)$. The so-called $\varepsilon$-closure $E(q)$ of a state $q$ of $N$ is given by

$$E(q) \;=\; \{\, \bar{q} \in Q_N \mid (q, \varepsilon) \;\vdash_N^*\; (\bar{q}, \varepsilon) \,\}$$

We construct a DFA $D = (Q_D,\, \Sigma,\, \delta,\, q_D^0, F_D)$ such that $\mathcal{L}(D) = \mathcal{L}(N)$ as follows.

- $Q_D = \mathcal{P}(Q_N)$, i.e. states of $D$ are sets of states of $N$;
- $\delta(Q, a) = \bigcup\{\, E(\bar{q}) \mid q \in Q,\; q \xrightarrow{a}_N \bar{q} \,\}$ for $Q \subseteq Q_N$;
- $q_D^0 = E(q_N^0)$, the $\varepsilon$-closure of the initial state of $N$;
- $F_D = \{\, Q \subseteq Q_N \mid Q \cap F_N \neq \emptyset \,\}$.

Thus, in particular, if $Q \xrightarrow{a}_D Q'$, i.e. if $\delta(Q, a) = Q'$, then we have, for all $q' \in Q_N$, that $q' \in Q'$ iff $q \xrightarrow{a}_N \bar{q}$ and $(\bar{q}, \varepsilon) \;\vdash_N^*\; (q', \varepsilon)$, for some $q \in Q$, $\bar{q} \in Q_N$. We claim

$$(q, w) \;\vdash_N^*\; (q', \varepsilon) \quad \text{iff} \quad \exists Q' \subseteq Q_N \colon (E(q), w) \;\vdash_D^*\; (Q', \varepsilon) \text{ and } q' \in Q'$$

Proof of the claim: ($\Rightarrow$) By induction on $|w|$. Basis, $|w| = 0$: Then we have $w = \varepsilon$. Thus $(q, \varepsilon) \;\vdash_N^*\; (q', \varepsilon)$, hence $q' \in E(q)$ by definition of $E(q)$. So $(E(q), w) \;\vdash_D^*\; (Q', \varepsilon)$. Induction step, $|w| > 0$: Then we have $w = va$ for suitable $v$ and $a$. Thus, for some $\bar{q}, \bar{q}' \in Q_N$,

$$(q, w) \;\vdash_N^*\; (\bar{q}, a) \;\vdash_N\; (\bar{q}', \varepsilon) \;\vdash_N^*\; (q', \varepsilon)$$

By Lemma 2.8, $(q, v) \;\vdash_N\; (\bar{q}, \varepsilon)$. Hence, by induction hypothesis, we can find $\bar{Q} \subseteq Q_N$ such that $(E(q), v) \;\vdash_D^*\; (\bar{Q}, \varepsilon)$ and $\bar{q} \in \bar{Q}$. By Lemma 2.3 we obtain $(E(q), w) \;\vdash_D^*\; (\bar{q}, a)$. Since $(\bar{q}', \varepsilon) \;\vdash_N^*\; (q', \varepsilon)$, we have $q' \in E(\bar{q}')$. Put $Q' = \bigcup\{\, E(\hat{q}') \mid \hat{q} \in \bar{Q},\; \hat{q} \xrightarrow{a}_N \hat{q}' \,\}$. Then $\bar{Q} \xrightarrow{a}_D Q'$. Note $\bar{q} \in \bar{Q}$ and $q \in Q'$. Combination of all this yields

$$(E(q), w) \;\vdash_D^*\; (\bar{Q}, a) \;\vdash_D\; (Q', \varepsilon) \quad \text{and} \quad q' \in Q'$$

Hence, $(E(q), w) \;\vdash_D^*\; (Q', \varepsilon)$ and $q' \in Q'$ as was to be shown.

($\Leftarrow$) By induction on $|w|$. Basis, $|w| = 0$: Then we have $w = \varepsilon$ and $Q' = E(q)$, since $D$ has no $\tau$-transitions. By definition, if $q' \in E(q)$, then $(q, \varepsilon) \;\vdash_N^*\; (q', \varepsilon)$. Induction step, $|w| > 0$: Then $w = va$ for suitable $v$ and $a$. It holds that

$$(E(q), w) \;\vdash_D^*\; (\bar{Q}, a) \;\vdash_D\; (Q', \varepsilon)$$

for some $\bar{Q} \subseteq Q_N$. Thus, again by Lemma 2.3, $(E(q), v) \;\vdash_D^*\; (\bar{Q}, \varepsilon)$. Since $(\bar{Q}, a) \;\vdash_D\; (Q', \varepsilon)$ and $q' \in Q'$, we have $\bar{q} \xrightarrow{a}_N \bar{q}'$ for some $\bar{q} \in \bar{Q}$ and $\bar{q}' \in Q_N$ such that $q' \in E(\bar{q}')$. By induction hypothesis, $(q, v) \;\vdash_N^*\; (\bar{q}, \varepsilon)$ and $(q, w) \;\vdash_N^*\; (\bar{q}, a)$ by Lemma 2.8. Moverover, $(\bar{q}, a) \;\vdash_N\; (\bar{q}', \varepsilon)$ and $(\bar{q}', \varepsilon) \;\vdash_N^*\; (q', \varepsilon)$. Combining all this yields

$$(q, w) \;\vdash_N^*\; (\bar{q}, a) \;\vdash_N\; (\bar{q}', \varepsilon) \;\vdash_N^*\; (q', \varepsilon)$$

Thus $(q, w) \;\vdash_N^*\; (q', \varepsilon)$ which proves the claim.

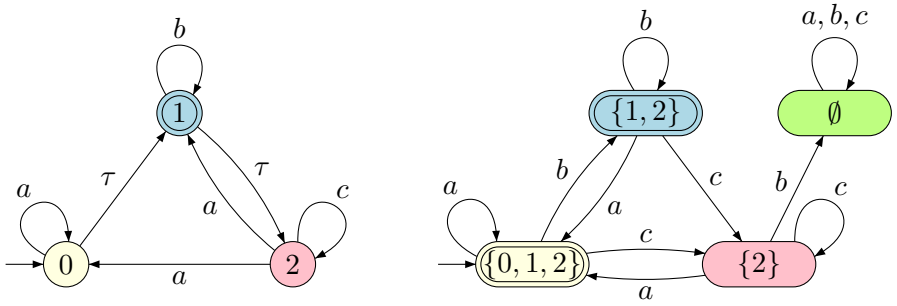



Figure 2.6: An NFA and a language equivalent DFA

Now, to show that $\mathcal{L}(N) = \mathcal{L}(D)$ we reason as follows:

$$
\begin{aligned}
& w \in \mathcal{L}(N) \\
\iff\ & \exists q' \in F_N \colon (q_N^0, w) \vdash_N^* (q', \varepsilon) \\
\iff\ & \exists q' \in F_N\, \exists Q' \subseteq Q_N \colon (E(q_N^0), w) \vdash_D^* (Q', \varepsilon) \wedge q' \in Q' \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by the claim)} \\
\iff\ & \exists Q' \in F_D \colon (E(q_N^0), w) \vdash_D^* (Q', \varepsilon) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(if } Q' \in F_D \text{, then } Q' \neq \emptyset\text{)} \\
\iff\ & w \in \mathcal{L}(D)
\end{aligned}
$$

This proves the theorem. □

The construction above, in the proof of Theorem 2.13, takes for $D$ the complete powerset $\mathcal{P}(Q_N)$ as set of states. Usually this leads to superfluous states, unreachable from the inital state. To end up with a smaller number of states in $D$, one can do this more cautiously.

**Example 2.14.** Consider the automaton $N$ depicted in Figure 2.6 for which we will construct a DFA $D$ accepting the same language. The set of states of $D$ will be built up lazily. Following the construction of Theorem 2.13 we need to include the starting state $E(0) = \{0, 1, 2\}$, the $\varepsilon$-closure of the starting state 0 of $N$.

We calculate possible transitions for $\{0, 1, 2\}$ for all symbols $a$, $b$, and $c$.

$$
\begin{aligned}
& 0 \xrightarrow{a}_N 0 \qquad E(0) = \{0, 1, 2\} \\
& 2 \xrightarrow{a}_N 0 \qquad E(0) = \{0, 1, 2\} \\
& 2 \xrightarrow{a}_N 1 \qquad E(1) = \{1, 2\} \\
& \text{thus } \{0, 1, 2\} \xrightarrow{a}_D \{0, 1, 2\} \text{ i.e. } \delta(\{0, 1, 2\}, a) = \{0, 1, 2\} \\
& 1 \xrightarrow{b}_N 1 \qquad E(1) = \{1, 2\} \\
& \text{thus } \{0, 1, 2\} \xrightarrow{b}_D \{1, 2\} \text{ i.e. } \delta(\{0, 1, 2\}, b) = \{1, 2\} \\
& 2 \xrightarrow{c}_N 2 \qquad E(2) = \{2\} \\
& \text{thus } \{0, 1, 2\} \xrightarrow{c}_D \{2\} \text{ i.e. } \delta(\{0, 1, 2\}, c) = \{2\}
\end{aligned}
$$

Note, apart from the initial state $\{0,1,2\}$ we have encountered two other states, viz. $\{1,2\}$ and $\{2\}$. We will first calculate the transitions for $\{1,2\}$.

$$\begin{aligned}
&2 \xrightarrow{a}_N 0 \qquad E(0) = \{0,1,2\}\\
&2 \xrightarrow{a}_N 1 \qquad E(1) = \{1,2\}\\
&\text{thus } \{1,2\} \xrightarrow{a}_D \{0,1,2\} \text{ i.e. } \delta(\{1,2\},a) = \{0,1,2\}\\
&1 \xrightarrow{b}_N 1 \qquad E(1) = \{1,2\}\\
&\text{thus } \{1,2\} \xrightarrow{b}_D \{1,2\} \text{ i.e. } \delta(\{1,2\},b) = \{1,2\}\\
&2 \xrightarrow{c}_N 2 \qquad E(2) = \{2\}\\
&\text{thus } \{1,2\} \xrightarrow{c}_D \{2\} \text{ i.e. } \delta(\{1,2\},c) = \{2\}
\end{aligned}$$

No new states have been introduced; we continue with calculating the transitions for state $\{2\}$.

$$\begin{aligned}
&2 \xrightarrow{a}_N 0 \qquad E(0) = \{0,1,2\}\\
&2 \xrightarrow{a}_N 1 \qquad E(1) = \{1,2\}\\
&\text{thus } \{2\} \xrightarrow{a}_D \{0,1,2\} \text{ i.e. } \delta(\{2\},a) = \{0,1,2\}\\
&2 \xrightarrow{c}_N 2 \qquad E(2) = \{2\}\\
&\text{thus } \{2\} \xrightarrow{c}_D \{2\} \text{ i.e. } \delta(\{2\},c) = \{2\}\\
&2 \text{ has no outgoing } b\text{-transition in } N\\
&\text{thus } \{2\} \xrightarrow{b}_D \emptyset \text{ i.e. } \delta(\{2\},b) = \emptyset
\end{aligned}$$

We choose as set of states $Q_D$ the states that have been introduced up to here. Thus,

$$Q_D \;=\; \{\, \{0,1,2\},\, \{1,2\},\, \{2\},\, \emptyset \,\}$$

This state is needed to obtain a complete transition function for $D$. Although the non-determinism of transitions has been resolved now, state $\{2\}$ is lacking a transition for $b$. So, we add $\{2\} \xrightarrow{b}_D \emptyset$ together with $\emptyset \xrightarrow{a}_D \emptyset$, $\emptyset \xrightarrow{b}_D \emptyset$, and $\emptyset \xrightarrow{c}_D \emptyset$. Put differently, we have $\delta(\{2\},b) = \emptyset$, $\delta(\emptyset,a) = \emptyset$, $\delta(\emptyset,b) = \emptyset$, and $\delta(\emptyset,c) = \emptyset$. The state $\emptyset$ is called a trap state. Once in, the automaton can't get out from there.

The final states of $D$ are those states in $Q_D$ that, as subsets of $Q_N$, contain 1, the single final state of $Q_N$. These are $\{0,1,2\}$ and $\{1,2\}$. The resulting DFA $D$ is depicted at the right of Figure 2.6.

## 2.3 Regular expressions

DFAs and NFAs are computational descriptions that are equivalent from a language perspective. In this section we introduce a syntactic alternative.

**Definition 2.15** (Regular expression)**.** Let $\Sigma$ be an alphabet. The class $RE_\Sigma$ of regular expressions over $\Sigma$ is defined as follows.

(i) $\mathbf{1}$ and $\mathbf{0}$ are regular expressions;

(ii) each $a \in \Sigma$ is a regular expression;

(iii) if $r_1$, $r_2$ and $r$ are regular expressions, then $(r_1 + r_2)$, $(r_1 \cdot r_2)$ and $(r^*)$ are regular expressions.

When clear from the context, the alphabet $\Sigma$ may be omitted as subscript of $RE_\Sigma$. For reasons to become clear in a minute, the regular expression $(r_1 + r_2)$ is called the union of $r_1$ and $r_2$, and the regular expression $(r_1 \cdot r_2)$ is called the concatenation of $r_1$ and $r_2$. The regular expression $(r^*)$ is called the Kleene-closure or iteration of $r$.

To reduce the number of parentheses we assume the $*$ construction to bind the most, followed by concatenation $\cdot$, and with choice $+$ having lowest priority. Typically, outermost parentheses will be suppressed too.

**Definition 2.16** (Language of a regular expression)**.** Let $RE_\Sigma$ be the class of regular expression over the alphabet $\Sigma$. The language $\mathcal{L}(r)$ of a regular expression $r$ is given by

(i) $\mathcal{L}(\mathbf{1}) = \{\varepsilon\}$ and $\mathcal{L}(\mathbf{0}) = \emptyset$;

(ii) $\mathcal{L}(a) = \{a\}$ for $a \in \Sigma$;

(iii) $\mathcal{L}(r_1 + r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$, $\mathcal{L}(r_1 \cdot r_2) = \mathcal{L}(r_1) \cdot \mathcal{L}(r_2)$, $\mathcal{L}(r^*) = \mathcal{L}(r)^*$.

In clause (iii), $\mathcal{L}(r_1)\mathcal{L}(r_2)$ denotes the concatenation of $\mathcal{L}(r_1)$ and $\mathcal{L}(r_2)$, while $\mathcal{L}(r)^*$ denotes the Kleene-closure of $\mathcal{L}(r)$. Recall

$$\begin{array}{rcl} L_1 \cdot L_2 & = & \{\, w_1 w_2 \mid w_1 \in L_1,\, w_2 \in L_2 \,\} \\ L^* & = & \{\, w_1 \cdots w_k \mid k \geqslant 0,\, w_1, \ldots, w_k \in L \,\} \end{array}$$

for languages $L_1$, $L_2$ and $L$.

**Example 2.17.** Consider with respect to the alphabet $\{a, b\}$ the regular expression $(a + b)^* \cdot a$. We have

$$\begin{array}{rcl} & & \mathcal{L}((a+b)^* \cdot a) \\ & = & \mathcal{L}((a+b)^*)\mathcal{L}(a) \\ & = & \mathcal{L}(a+b)^*\{a\} \\ & = & (\mathcal{L}(a) \cup \mathcal{L}(b))^*\{a\} \\ & = & (\{a\} \cup \{b\})^*\{a\} \\ & = & \{a, b\}^*\{a\} \\ & = & \{\, w \in \{a, b\}^* \mid w \text{ ends with } a \,\} \end{array}$$

**Example 2.18.** The set of strings over the alphabet $\{a, b, c\}$ that contain at least one $a$ and at most one $b$ can be represented by the regular expression

$$(\, c^* \cdot a \cdot (a + c)^* \cdot (b + \mathbf{1}) \cdot (a + c)^* \,) + (\, c^* \cdot (b + \mathbf{1}) \cdot c^* \cdot a \cdot (a + c)^* \,)$$

Often we write $a_1a_2\cdots a_n$ rather than $a_1 \cdot a_2 \cdot \cdots \cdot a_n$ using juxtaposition rather than concatenation.

**Theorem 2.19.** If a language $L$ is accepted by a DFA then $L$ is the language of a regular expression.

*Proof.* Suppose $L \subseteq \Sigma^*$ and $L = \mathcal{L}(D)$ for a deterministic finite automaton $D = (Q, \Sigma, \delta, q_1, F)$ with $Q = \{ q_1, q_2, \ldots, q_n \}$. Note the start state $q_1$.

Let the regular expression $R^k_{i,j}$ represent the set of strings $w$ that label the paths from state $q_i$ to state $q_j$ without passing through a state $q_\ell$ with $\ell > k$ in between. More precisely, we define $R^k_{i,j}$, for $i, j = 1, \ldots, n$, $k = 0, \ldots, n$, by

$$\begin{aligned} R^0_{i,i} &= a^1_{i,i} + \cdots + a^{s(i,i)}_{i,i} + \mathbf{1} \\ R^0_{i,j} &= a^1_{i,j} + \cdots + a^{s(i,j)}_{i,j} \qquad \text{for } i \neq j \\ R^{k+1}_{i,j} &= R^k_{i,j} + R^k_{i,k+1} \cdot ( R^k_{k+1,k+1} )^* \cdot R^k_{k+1,j} \end{aligned}$$

where $a^1_{i,j}, \ldots, a^{s(i,j)}_{i,j}$ are all symbols in $\Sigma$ such that $\delta(q_i, a^1_{i,j}) = q_j$, $\ldots$, $\delta(q_i, a^{s(i,j)}_{i,j}) = q_j$, for $i, j = 1, \ldots, n$.

We verify that the regular expressions $R^k_{i,j}$ are as intended: If $k = 0$ then the path from $q_i$ to $q_i$ has no intermediate states. So, either the path has length 1 and consists of one transition $q_i \xrightarrow{a}_D q_i$ yielding the subexpression $a$ with meaning $\{a\}$, or the path has length 0, yielding the subexpression $\mathbf{1}$ with meaning $\{\varepsilon\}$. This explains $R^0_{i,i}$. The explanation of the regular expression $R^0_{i,j}$, for $i \neq j$ is similar. However, in this case there is no empty path from $q_i$ to $q_j$, so no subexpression $\mathbf{1}$ either. With respect to $R^{k+1}_{i,j}$, a path from $q_i$ to $q_j$ which may pass through intermediate states $q_1, \ldots, q_{k+1}$, may or may not pass through state $q_{k+1}$. If not, the path is already represented by $R^k_{i,j}$. If the path passes through $q_{k+1}$, there is an initial part from $q_i$ to $q_{k+1}$ and a final part from $q_{k+1}$ to $q_j$ that both avoid $q_{k+1}$. In between the path can be split up in subpaths from $q_{k+1}$ to $q_{k+1}$ that do not pass through $q_{k+1}$ as intermediate state; it may pass through intermediate states $q_1$ to $q_k$ only. The initial part and final part are captured by $R^k_{i,k+1}$ and $R^k_{k+1,j}$, respectively. The subpaths from $q_{k+1}$ to $q_{k+1}$ with intermediate states from $q_1$ to $q_k$ are included in $R^k_{k+1,k+1}$. Any repetition of such path is allowed, which explains the Kleene star.

We have, for a string $w$ over the alphabet $\Sigma$, $w \in \mathcal{L}(D)$ iff there exists a path labeled $w$ from the initial state $q_1$ of $D$ to a final state $q_f \in F$ iff $w \in \mathcal{L}(R^n_{1,f})$ for a final state $q_f$. Thus, if $F = \{q_{f_1}, \ldots, q_{f_m}\}$, then $w \in \mathcal{L}(D)$ iff $w \in \mathcal{L}( R^n_{1,q_{f_1}} + \cdots R^n_{1,q_{f_m}} )$. This proves the theorem. □

**Example 2.20.** Consider the DFA $D$ given in Figure 2.7. The DFA has three states.

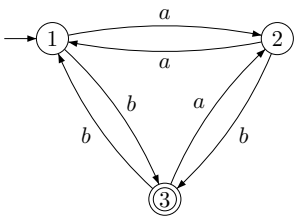



Figure 2.7: DFA of Example 2.20

We compute the regular expressions $R^k_{i,j}$ for $1 \leqslant i, j \leqslant 3$, $0 \leqslant k \leqslant 3$. We have

| $R^0_{i,j}$ | 1 | 2 | 3 |
|---|---|---|---|
| 1 | $\mathbf{1}$ | $a$ | $b$ |
| 2 | $a$ | $\mathbf{1}$ | $b$ |
| 3 | $b$ | $a$ | $\mathbf{1}$ |

| $R^1_{i,j}$ | 1 | 2 | 3 |
|---|---|---|---|
| 1 | $\mathbf{1}$ | $a$ | $b$ |
| 2 | $a$ | $\mathbf{1} + aa$ | $b + ab$ |
| 3 | $b$ | $a + ba$ | $\mathbf{1} + bb$ |

In order to avoid too much unnecessary computation we reason as follows: We need to compute $R^3_{1,3}$, since $q_1$ is the initial state, $q_3$ is the only final state, and there are three states in total. By definition

$$\begin{aligned} R^3_{1,3} &= R^2_{1,3} + R^2_{1,3} \cdot (R^2_{3,3})^* \cdot R^2_{3,3} \\ R^2_{1,3} &= R^1_{1,3} + R^1_{1,2} \cdot (R^1_{2,2})^* \cdot R^1_{2,3} \\ R^2_{3,3} &= R^1_{3,3} + R^1_{3,2} \cdot (R^1_{2,2})^* \cdot R^1_{2,3} \end{aligned}$$

Using the table of results for $R^1_{i,j}$ to compute $R^2_{1,3}$ and $R^2_{3,3}$ we get

$$\begin{aligned} R^2_{1,3} &= b \,+\, a \cdot (\mathbf{1} + aa)^* \cdot (b + ab) \\ &= b \,+\, a \cdot (aa)^* \cdot (b + ab) \\ R^2_{3,3} &= (\mathbf{1} + bb) \,+\, (a + ba) \cdot (\mathbf{1} + aa)^* \cdot (b + ab) \\ &= \mathbf{1} \,+\, bb \,+\, (a + ba) \cdot (aa)^* \cdot (b + ab) \end{aligned}$$

Therefore

$$\begin{aligned} R^3_{1,3} &= \big(b \,+\, a \cdot (aa)^* \cdot (b + ab)\big) \,+ \\ &\quad \big(b \,+\, a \cdot (aa)^* \cdot (b + ab)\big) \cdot \\ &\qquad \big(\mathbf{1} \,+\, bb \,+\, (a + ba) \cdot (aa)^* \cdot (b + ab)\big)^* \cdot \\ &\qquad\quad \big(\mathbf{1} \,+\, bb \,+\, (a + ba) \cdot (aa)^* \cdot (b + ab)\big) \\ &= \big(\, b \,+\, a \cdot (aa)^* \cdot (b + ab)\,\big) \cdot \big(\, bb \,+\, (a + ba) \cdot (aa)^* \cdot (b + ab)\,\big)^* \end{aligned}$$

Above we have used the many laws for regular expressions, e.g. the equalities $r_1 + r_1 \cdot r_2^* = r_1 \cdot r_2^*$, $(1 + r)^* = r^*$ and $r^* \cdot (\mathbf{1} + r) = r^*$ that hold for arbitrary regular expressions $r_1$, $r_2$ and $r$.

Another way to obtain a regular expression from a DFA is using systems of equations. For a concise description of the equations we identify a regular expression $r$ and its language $\mathcal{L}(r)$. So, we may write $L = r_1 \cdot L + r_2$ replacing $L = \mathcal{L}(r_1) \cdot L + \mathcal{L}(r_2)$. The key observation is that, for arbitrary regular expressions $r_1$ and $r_2$, there exists exactly one language $L$ such that $L = r_1 \cdot L + r_2$, namely $L = \mathcal{L}(r_1^* \cdot r_2)$. Thus, if $L$ satisfies $L = r_1 \cdot L + r_2$, then $L = \mathcal{L}(r_1^* \cdot r_2)$.

**Example 2.21.** The DFA $D$ of Figure 2.7 can be represented by the system of equations

$$\begin{array}{rcl} X & = & a \cdot Y + b \cdot Z \\ Y & = & a \cdot X + b \cdot Z \\ Z & = & b \cdot X + a \cdot Y + \mathbf{1} \end{array}$$

Here, the variable $X$ corresponds to state 1, variable $Y$ to state 2, and variable $Z$ to state 3. For example, the equation for $X$ has the summand $a.Y$, because the DFA has a transition from state 1 to state 2 labeled $a$. Note the summand $\mathbf{1}$ in the equation for $Z$. This reflects that state 3 is a final state. Each equation represents the language belonging to the corresponding state. In particular, since $X$ represents the initial state of the automaton, the language of $X$ coincides with the language of the DFA.

Next we want to solve the system. We do this by elimination of variables. First we eliminate $Z$ from the equations of $X$ and $Y$. At the right-hand side of the equation of $Y$ an expression with $X$ and $Y$ remains. We solve for $Y$ using the above observation. Then we eliminate $Y$ from the resulting equation for $X$. This yields an equation with $X$ only, that we now solve for $X$, again making use of the above mentioned principle.

As in the third equation above the variable $Z$ is expressed in the variables $X$ and $Y$, we first substitute the right-hand side of $Z$ in the equations for $X$ and $Y$. This yields

$$\begin{array}{rcl} X & = & a \cdot Y + b \cdot (b \cdot X + a \cdot Y + \mathbf{1}) \\ Y & = & a \cdot X + b \cdot (b \cdot X + a \cdot Y + \mathbf{1}) \end{array}$$

and after application of some simplifying rules we obtain

$$\begin{array}{rcl} X & = & b \cdot b \cdot X + (a + b \cdot a) \cdot Y + b \cdot \mathbf{1} \\ Y & = & (a + b \cdot b) \cdot X + b \cdot a \cdot Y + b \cdot \mathbf{1} \\ & = & b \cdot a \cdot Y + (a + b \cdot b) \cdot X + b \cdot \mathbf{1} \end{array}$$

Solving the equation for $Y$ results in

$$Y = (b \cdot a)^* \cdot \big((a + b \cdot b) \cdot X + b \cdot \mathbf{1}\big)$$

When we eliminate $Y$ from the equation of $X$ we obtain

$$\begin{array}{rcl} X & = & b \cdot b \cdot X + (a + b \cdot a) \cdot \Big((b \cdot a)^* \cdot \big((a + b \cdot b) \cdot X + b \cdot \mathbf{1}\big)\Big) + b \cdot \mathbf{1} \\ & = & \big(b \cdot b + (a + b \cdot a) \cdot (b \cdot a)^* \cdot (a + b \cdot b)\big) \cdot X + (a + b \cdot a) \cdot b \cdot \mathbf{1} + b \cdot \mathbf{1} \end{array}$$

Thus, again using the key observation above, we obtain

$$X = \big(b \cdot b + (a + b \cdot a) \cdot (b \cdot a)^* \cdot (a + b \cdot b)\big)^* \cdot \big((a + b \cdot a) \cdot b \cdot \mathbf{1} + b \cdot \mathbf{1}\big)$$

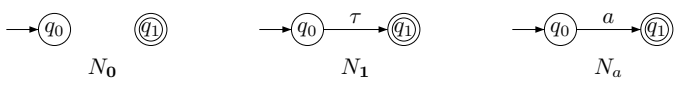



Figure 2.8: NFAs for regular expressions $\mathbf{0}$, $\mathbf{1}$ and $a$

and we conclude

$$\begin{aligned}\mathcal{L}(D) &= \big(b\cdot b + (a+b\cdot a)\cdot(b\cdot a)^*\cdot(a+b\cdot b)\big)^*\cdot\big((a+b\cdot a)\cdot b\cdot\mathbf{1}+b\cdot\mathbf{1}\big)\\ &= \big(bb+(a+ba)\cdot(ba)^*\cdot(a+bb)\big)^*\cdot(ab+bab+b)\end{aligned}$$

**Theorem 2.22.** If a language $L$ equals $\mathcal{L}(r)$ for some regular expression $r$, then $L$ equals $\mathcal{L}(N)$ for some NFA $N$.

*Proof.* Suppose $L \subseteq \Sigma^*$. We prove the theorem by structural induction on the expression $r$. For each regular expression $r$ we will construct an NFA $N_r$ such that $\mathcal{L}(N_r) = \mathcal{L}(r)$. Moreover, we take care that

(i) $N_r$ has exactly one final state;

(ii) the inital state of $N_r$ has only outgoing transitions (if any);

(iii) the final state of $N_r$ has only incoming transitions.

We need to distinguish three base cases and three successor cases.

Basis, $r = \mathbf{0}$: We have $\mathcal{L}(\mathbf{0}) = \emptyset$. Clearly, $\mathcal{L}(\mathbf{0}) = \mathcal{L}(N_{\mathbf{0}})$ for $N_{\mathbf{0}}$ depicted at the left of Figure 2.8.

Basis, $r = \mathbf{1}$: We have $\mathcal{L}(\mathbf{1}) = \{\varepsilon\}$. Clearly, $\mathcal{L}(\mathbf{1}) = \mathcal{L}(N_{\mathbf{1}})$ for $N_{\mathbf{1}}$ depicted in the center of Figure 2.8.

Basis, $r = a$ for $a \in \Sigma$: We have $\mathcal{L}(a) = \{a\}$. Clearly, $\mathcal{L}(a) = \mathcal{L}(N_a)$ for $N_a$ depicted at the right of Figure 2.8.

Induction step, $r = r_1 + r_2$: Suppose NFA $N_i$ accepts the language $\mathcal{L}(r_i)$, say $N_i = (Q_i, \Sigma, \delta_i, q_0^i, \{q_f^i\})$, for $i = 1, 2$. Moverover, we assume $Q_1 \cap Q_2 = \emptyset$. Let $q_0$ and $q_f$ be two fresh states. Then we put $N_r = (Q, \Sigma, \delta, q_0, \{q_f\})$ where $Q = Q_1 \cup Q_2 \cup \{q_0, q_f\}$ and $\delta$ extends $\delta_1$ and $\delta_2$ with $q_0 \xrightarrow{\tau} q_0^1$, $q_0 \xrightarrow{\tau} q_0^2$ and $q_f^1 \xrightarrow{\tau} q_f$, $q_f^2 \xrightarrow{\tau} q_f$. Clearly, by construction, $\mathcal{L}(N_r) = \mathcal{L}(N_1) \cup \mathcal{L}(N_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2) = \mathcal{L}(r_1 + r_2)$. See the left-upper part of Figure 2.9.

Induction step, $r = r_1 \cdot r_2$: Suppose NFA $N_i$ accepts the language $\mathcal{L}(r_i)$, say $N_i = (Q_i, \Sigma, \delta_i, q_0^i, \{q_f^i\})$, for $i = 1, 2$. Again, we assume $Q_1 \cap Q_2 = \emptyset$. We put $N_r = (Q_1 \cup Q_2, \Sigma, \delta, q_0^1, \{q_f^2\})$ where $\delta$ extends $\delta_1$ and $\delta_2$ with $q_f^1 \xrightarrow{\varepsilon} q_0^2$. Clearly, by construction, $\mathcal{L}(N_r) = \mathcal{L}(N_1)\cdot\mathcal{L}(N_2) = \mathcal{L}(r_1)\cdot\mathcal{L}(r_2) = \mathcal{L}(r_1\cdot r_2)$. See the right-upper part of Figure 2.9.

Induction step, $r = r_0^*$: Suppose NFA $N_0$ accepts the language $\mathcal{L}(r_0)$, say $N_0 = (Q_0, \Sigma, \delta_0, q_0^0, \{q_f^0\})$. Let $q_0$ and $q_f$ be two fresh states. Then we put $N_r = (Q_0 \cup$

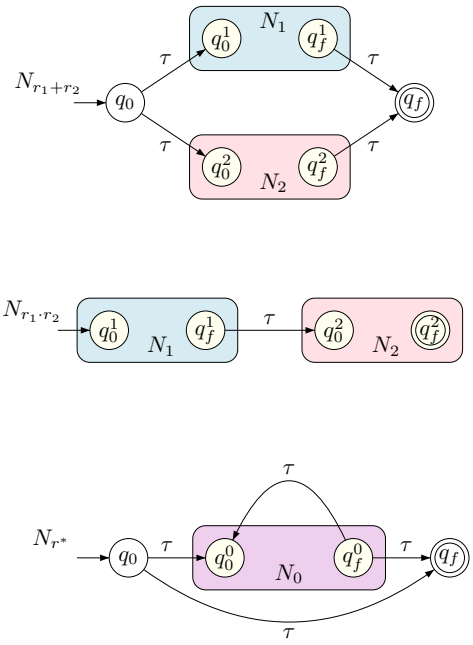



Figure 2.9: NFAs for regular expressions $r_1 + r_2$, $r_1 \cdot r_2$ and $r^*$

$\{q_0, q_f\}$, $\Sigma$, $\delta$, $q_0, \{q_f\}$) where $\delta$ extends $\delta_0$ with $q_0 \xrightarrow{\tau} q_0^0$, $q_0 \xrightarrow{\tau} q_f$, and $q_f^0 \xrightarrow{\tau} q_0^0$. Clearly, by construction, $\mathcal{L}(N_r) = \mathcal{L}(N_0)^* = \mathcal{L}(r_0)^* = \mathcal{L}(r_0^*)$. See the lower part of Figure 2.9. □

**Example 2.23.** Figure 2.10 depicts the NFA obtained by the construction described in the proof of Theorem 2.22 for the regular expression $(a + b)^* \cdot b \cdot (a + b)$.

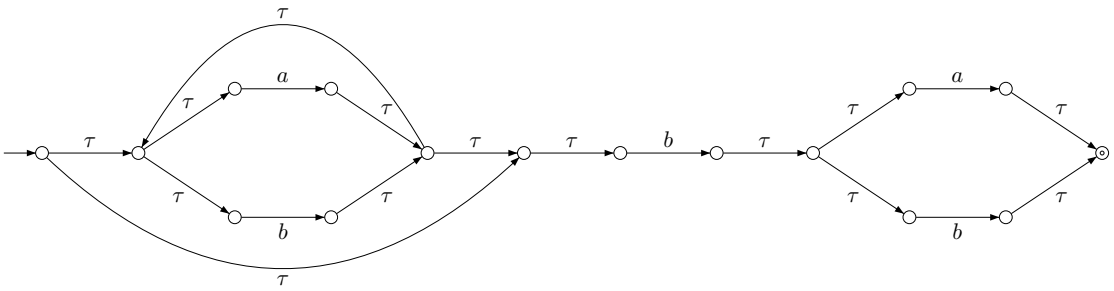



Figure 2.10: NFA of Example 2.23

## 2.4 Properties of the class of regular languages

In this section we further relate the concepts of a DFA, an NFA and a regular expression. It turns out that they accept the same class of languages, the class of regular languages. We provide a means to prove that a language is not regular, look at closure properties and decision procedures for the class of regular languages.

**Definition 2.24.** A language $L \subseteq \Sigma^*$ over an alphabet $\Sigma$ is called a regular language if there exists an NFA $N$ such that $L = \mathcal{L}(N)$.

The theorem states that NFAs, DFAs and regular expressions define the same class of languages, viz. the class of regular languages. The theorem, or rather the theorems its proof refers to, also provides flexibility in the representation of a regular language. Given a regular language $L$ as either the language accepted by an NFA or a DFA or as the language of a regular expression, we can construct an NFA or a DFA that accepts $L$ or a regular expression who's language is $L$.

**Theorem 2.25.** Let $L$ be a language. The following three statements are equivalent:

(i) $L$ is a regular language

(ii) There exists a DFA $D$ such that $L = \mathcal{L}(D)$.

(iii) There exists a regular expression $r$ such that $L = \mathcal{L}(r)$.

$(i) \Rightarrow (ii)$. By definition there exists an NFA $N$ such that $L = \mathcal{L}(N)$. By Theorem 2.13 there exists a DFA $D$ such that $\mathcal{L}(N) = \mathcal{L}(D)$. Then, clearly, $L = \mathcal{L}(D)$ for the DFA $D$.

$[(ii) \Rightarrow (iii)]$ Suppose $L = \mathcal{L}(D)$ for a DFA $D$. By Theorem 2.19 there exists a regular expresion $r$ such that $\mathcal{L}(D) = \mathcal{L}(r)$. Thus clearly, $L = \mathcal{L}(r)$ for the regular expression $r$.

$[(iii) \Rightarrow (i)]$ Suppose $L = \mathcal{L}(r)$ for a regular expression $r$. By Theorem 2.22 exists an NFA $N$ such that $\mathcal{L}(N) = \mathcal{L}(r)$. Therefore $L = \mathcal{L}(N)$ and $L$ is a regular language. □

Next we investigate closure properties of the class of regular languages. The following theorem states that the class of regular languages is closed under union, complement and intersection.

**Theorem 2.26.**

(a) If $L_1, L_2$ are regular languages, then $L_1 \cup L_2$ is a regular language too.

(b) If the language $L \subseteq \Sigma^*$ is regular, then so is $L^C = \Sigma^* \setminus L$.

(c) If $L_1, L_2$ are regular languages, then $L_1 \cap L_2$ is a regular language too.

*Proof.* (a) By Theorem 2.19 we can find regular expressions $r_1$ and $r_2$ such that $L_1 = \mathcal{L}(r_1)$ and $L_2 = \mathcal{L}(r_2)$. Then we have $L_1 \cup L_2 = \mathcal{L}(r_1) \cup \mathcal{L}(r_2) = \mathcal{L}(r_1 + r_2)$. Thus, by Theorem 2.25 $L_1 \cup L_2$ is a regular language.

(b) Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA such that $\mathcal{L}(D) = L$. Define the DFA $D' = (Q, \Sigma, \delta, q_0, F')$ by putting $F' = Q \setminus F$. Thus $q \in Q$ is a final state in $D'$ iff $q$ is not a final state in $D$. With appeal to Lemma 2.3 we have $w \in \mathcal{L}(D')$ iff $((q_0, w) \vdash^*_{D'} (q, \varepsilon)$ and $q \in F')$ iff $((q_0, w) \vdash^*_D (q, \varepsilon)$ and $q \notin F)$ iff $w \notin \mathcal{L}(D)$.

(c) By the laws of De Morgan, $L_1 \cap L_2 = (L_1^C \cup L_2^C)^C$. The languages $L_1^C$ and $L_2^C$ are regular, by regularity of $L_1$ and $L_2$ and Theorem 2.26. Thus $L_1^C \cup L_2^C$ is regular, by Theorem 2.26. Therefore, $(L_1^C \cup L_2^C)^C$ is regular, again by Theorem 2.26. □

Note, for item (b), it is important that the automaton we consider is deterministic. In general, changing acceptance and non-acceptance in an NFA does not lead to the complement of the accepted language.

Not every language is a regular language. Put differently, not every language can be accepted by an NFA. The next theorem, aptly called the Pumping Lemma, can be used to prove a language not to be regular.

**Theorem 2.27** (Pumping Lemma for regular languages)**.** Let $L$ be a regular language over an alphabet $\Sigma$. There exists a constant $m > 0$ such that each $w \in L$ with $|w| \geqslant m$ can be written as $w = xyz$ where $x, y, z \in \Sigma^*$, $y \neq \varepsilon$, $|xy| \leqslant m$, and for all $k \geqslant 0$: $xy^k z \in L$.

*Proof.* Suppose $L = \mathcal{L}(D)$ for a DFA $D$. Choose $m$ to be the number of states of $D$. Let $w \in L$ with $|w| \geqslant m$, say $w = a_1 \cdots a_n$, thus $n \geqslant m$. Pick $n + 1$ states $q_0, \ldots, q_n$ with $q_0$ the initial state of $D$, $q_{i-1} \xrightarrow{a_i}_D q_i$, for $1 \leqslant i \leqslant n$, and $q_n$ a final state. Thus $(q_{i-1}, a_i a_{i+1} \cdots a_n) \vdash_D (q_i, a_{i+1} \cdots a_n)$. Since $D$ has $m$ states, the states $q_0, \ldots, q_m$ cannot all be different. Pick $m_1, m_2$ such that $0 \leqslant m_1 < m_2 \leqslant m$ and $q_{m_1} = q_{m_2}$. Put

$$x = a_1 \cdots a_{m_1} \qquad y = a_{m_1+1} \cdots a_{m_2} \qquad z = a_{m_2+1} \cdots a_n$$

We have $xyz = a_1 \cdots a_{m_1} a_{m_1+1} \cdots a_{m_2} a_{m_2+1} \cdots a_n = w$, $y \neq \varepsilon$ since $m_1 < m_2$, and $|xy| \leqslant m$ since $m_2 \leqslant m$.

We verify that $xy^k z \in L$ for all $k \geqslant 0$: It holds that

$$(q_0, x) \vdash^*_D (q_{m_1}, \varepsilon) \qquad (q_{m_1}, y) \vdash^*_D (q_{m_2}, \varepsilon) \qquad (q_{m_2}, z) \vdash^*_D (q_n, \varepsilon)$$

Since $q_{m_1} = q_{m_2}$, it follows that $(q_{m_1}, y) \vdash^*_D (q_{m_1}, \varepsilon)$, thus $(q_{m_1}, y^k) \vdash^*_D (q_{m_1}, \varepsilon)$ and $(q_{m_1}, y^k) \vdash^*_D (q_{m_2}, \varepsilon)$ for arbitrary $k \geqslant 0$. Therefore

$$(q_0, x) \vdash^*_D (q_{m_1}, \varepsilon) \qquad (q_{m_1}, y^k) \vdash^*_D (q_{m_2}, \varepsilon) \qquad (q_{m_2}, z) \vdash^*_D (q_n, \varepsilon)$$

for arbitrary $k \geqslant 0$, and hence $(q_0, xy^k z) \vdash^*_D (q_n, \varepsilon)$. Thus $w = xy^k z \in \mathcal{L}(D) = L$ since $q_n$ is a final state, as was to be shown. □

**Example 2.28.** The language $L = \{ a^n b^n \mid n \geqslant 0 \}$ is not a regular language: Let $m > 0$ and consider $w = a^m b^m \in L$. Suppose we can split $w = xyz$ such that $y \neq \varepsilon$, $|xy| \leqslant m$, and $xy^k z \in L$ for all $k \geqslant 0$. Then the string $y$ is a non-empty string of $a$'s, say $y = a^\ell$. Thus $w' = xy^2 z = a^{m+\ell} b^m$, but $w' \notin L$. We conclude that there is no constant $m$ as mentioned by the Pumping Lemma, and therefore $L$ is not a regular language.

**Example 2.29.** The language $L = \{ a^{n^2} \mid n \geqslant 0 \}$ is not a regular language: Let $m > 0$ and consider $w = a^{m^2} \in L$. Suppose we can split $w = xyz$ such that $y \neq \varepsilon$, $|xy| \leqslant m$, and $xy^kz \in L$ for all $k \geqslant 0$. Then the string $y$ is a non-empty string of $a$'s, say $y = a^\ell$ with $1 \leqslant \ell \leqslant m$. Put $w' = xy^2z$, then we have $w' = a^{m^2+\ell}$. But $m^2 < m^2 + \ell \leqslant m^2 + m < m^2 + 2m + 1 = (m+1)^2$, thus $m^2 + \ell$ isn't a square. So, $w' \notin L$. Thus, we conclude that there is no constant $m$ as mentioned by the Pumping Lemma, and therefore $L$ is not a regular language.

We close the chapter by looking into two decision algorithms for regular languages. The first decision algorithm needs to decide whether given a *regular* language $L$ as input, $L$ is or is not the empty language. We first solve the question if $L$ were be given by a regular expression. With appeal to Theorem 2.25 we can conclude that the theorem holds as well for $L$ given as the language accepted by an NFA.

**Theorem 2.30.** Let $L$ be a regular language over an alphabet $\Sigma$ represented by an NFA $N$ accepting $L$. Then it can be decided if $L = \emptyset$ or not.

*Proof.* We decide, for a regular expression $r$, emptiness of $L(r)$ as follows:

- $L = \emptyset$ if $r = \mathbf{0}$;
- $L \neq \emptyset$ if $r = \mathbf{1}$;
- $L \neq \emptyset$ if $r = a$ for some $a \in \Sigma$;
- $L = \emptyset$ if $r = r_1 + r_2$ for two regular expressions $r_1$ and $r_2$ and both $\mathcal{L}(r_1)$ and $\mathcal{L}(r_2)$ are empty, $L \neq \emptyset$ if $\mathcal{L}(r_1)$ or $\mathcal{L}(r_2)$ is not-empty;
- $L = \emptyset$ if $r = r_1 \cdot r_2$ for two regular expressions $r_1$ and $r_2$ and either $\mathcal{L}(r_1)$ or $\mathcal{L}(r_2)$ is empty, $L \neq \emptyset$ if both $\mathcal{L}(r_1)$ and $\mathcal{L}(r_2)$ are not-empty;
- $L \neq \emptyset$ if $r = (r')^*$ for some regular expression $r'$.

Note that the decision procedure terminates since the recusive calls for $r_1$, $r_2$ and $r'$ above involve a structurally simpler argument.

Now, suppose $L = \mathcal{L}(N)$ for an NFA $N$. Construct, using the algorithms given in the proofs of Theorem 2.13 and Theorem 2.19, a regular expression $r$ such that $L = \mathcal{L}(r)$ and decide whether $\mathcal{L}(r) = \emptyset$. □

Finally we consider a decision algorithm for membership. Given a regular language $L \subseteq \Sigma^*$ and a string $w \in \Sigma^*$, is it the case or is it not the case that $w \in L$?

**Theorem 2.31.** Let $L \subseteq \Sigma^*$ be a regular language over the alphabet $\Sigma$, represented by an NFA $N$ accepting $L$, and let $w \in \Sigma^*$ be a string over $\Sigma$. Then it can be decided if $w \in L$ or not.

*Proof.* Construct, using the algorithm given in the proof of Theorem 2.13, a DFA $D$ such that $\mathcal{L}(D) = \mathcal{L}(N)$. Simulate $D$ starting from its initial state on input $w$, say $(q_0, w) \vdash_D^* (q', \varepsilon)$ for some state $q'$ of $D$. Decide $w \in L$ if $q'$ is a final state of $D$, decide $w \notin L$ otherwise. □

# Chapter 3

# Push-Down Automata and Context-Free Languages

In the previous chapter, we studied finite automata, modeling computers without memory. In the next chapter, we study a general model of computers with memory. In the current chapter, we study an interesting class that is in between: a class of automata with a memory in the form of a stack, so-called *push-down automata*. The class of languages associated with push-down automata is often called the class of *context-free* languages. This class is important in the study of programming languages, in particular in parsing with context-free grammars.

## 3.1 Push-down automata

We consider an abstract model of computer with a memory in the form of a stack. This stack can be accessed only at the top: something can be added on top of the stack (push), or something can be removed from the top of the stack (pop). See Figure 3.1 for the architecture of a push-down automaton. From the start we allow internal steps $\tau$, that consume no input symbol but can modify the stack.

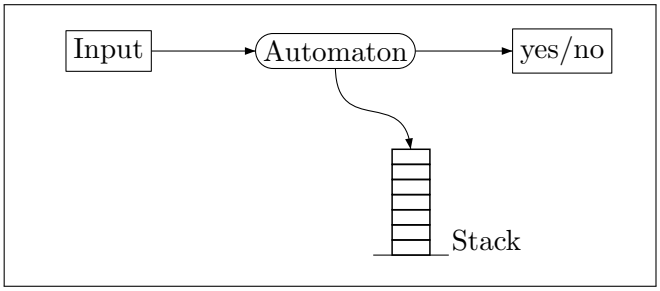



Figure 3.1: Architecture of a push-down automaton.

**Example 3.1.** Let us look at an example of a push-down automaton. In Figure 3.2, we indicate the initial state $q_0$ as usual. There, the stack is assumed to be empty, signaled by the special symbol $\emptyset$. From this state, we can input $a$, thereby replacing the empty stack by the stack containing a single data element 1 while control switches from state $q_0$ to state $q_1$. In the state $q_1$, we can either input $a$ again, replacing the top 1 by 11, and remain in state $q_1$, or alternatively input $b$, popping the top 1, putting nothing in its place, going to state $q_2$. There, we can read in $b$ repeatedly, popping 1's, but we must terminate and go to final state $q_3$ if (and only if) the stack has become empty.

We see that this push-down automaton can read any number of $a$'s (collecting exactly so many 1's on the stack), followed by the input of the same number of $b$'s followed by termination. As we shall see, its language is $\{ a^n b^n \mid n \geqslant 1 \}$, a non-regular language.

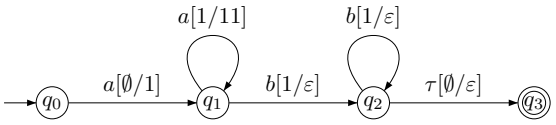



Figure 3.2: Example push-down automaton.

**Definition 3.2** (Push-down automaton)**.** A *push-down automaton* (PDA) is a septuple $P = (\, Q,\, \Sigma,\, \Delta,\, \emptyset,\, \to,\, q_0,\, F\,)$ where

1. $Q$ is a finite set of states,
2. $\Sigma$ is a finite input alphabet with $\tau \notin \Sigma$,
3. $\Delta$ is a finite data alphabet or stack alphabet,
4. $\emptyset \notin \Delta$ a special symbol denoting an empty stack,
5. $\to \subseteq Q \times \Sigma_\tau \times \Delta_\emptyset \times \Delta^* \times Q$, where $\Sigma_\tau = \Sigma \cup \{\tau\}$ and $\Delta_\emptyset = \Delta \cup \{\emptyset\}$, is a finite set of *transitions* or *steps*,
6. $q_0 \in Q$ is the initial state,
7. $F \subseteq Q$ is the set of final states.

We use $x$ and $y$ for strings in $\Delta^*$ denoting the content of a stack. If $(q, a, d, x, q') \in \to$ with $d \in \Delta$, we write $q \xrightarrow{a[d/x]} q'$, and this means that the machine, when it is in state $q$ and $d$ is the top element of the stack, can consume input symbol $a$, replace $d$ by the string $x$ and thereby move to state $q'$. Likewise, if $(q, a, \emptyset, x, q') \in \to$ we write $q \xrightarrow{a[\emptyset/x]} q'$ meaning that the machine, when it is in state $q$ and the stack is empty, can consume input symbol $a$, put the string $x$ on the stack and thereby move to state $q'$. In steps $q \xrightarrow{\tau[d/x]} q'$ and $q \xrightarrow{\tau[\emptyset/x]} q'$, no input symbol is consumed, only the stack is modified.

Note the occurrence of $\Delta_\emptyset$ vs. $\Delta^*$ for the transition relation $\rightarrow$. For a transition of the form $q \xrightarrow{a[\emptyset/\varepsilon]} q'$, the symbol $a$ is read from input but the stack remains empty.

**Example 3.3.** Consider again Figure 3.2. The figure depicts a push-down automaton $P = (\, Q,\, \Sigma,\, \Delta,\, \emptyset,\, \rightarrow,\, q_0,\, \{q_3\}\,)$ with $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, and $\Delta = \{1\}$ as set of states, input alphabet and stack alphabet, respectively, transitions

$$q_0 \xrightarrow{a[\emptyset/1]} q_1, \quad q_1 \xrightarrow{a[1/11]} q_1, \quad q_1 \xrightarrow{b[1/\varepsilon]} q_2, \quad q_2 \xrightarrow{b[1/\varepsilon]} q_2, \quad q_2 \xrightarrow{\tau[\emptyset/\varepsilon]} q_3$$

initial state $q_0$, and the singleton $\{q_3\}$ as the set of final states.

**Definition 3.4** (Configuration)**.** Let $P = (\, Q,\, \Sigma,\, \Delta,\, \emptyset,\, \rightarrow,\, q_0, F\,)$ be a push-down automaton. A configuration or instantaneous description (ID) of $P$ is a triple $(q, w, x) \in Q \times \Sigma^* \times \Delta^*$. The relation $\vdash_P \subseteq (\, Q \times \Sigma^* \times \Delta^*\,) \times (\, Q \times \Sigma^* \times \Delta^*\,)$ is given as follows.

(i) $(q, aw, dy) \vdash_P (p, w, xy)$ if $q \xrightarrow{a[d/x]} p$

(ii) $(q, w, dy) \vdash_P (p, w, xy)$ if $q \xrightarrow{\tau[d/x]} p$

(iii) $(q, aw, \varepsilon) \vdash_P (p, w, x)$ if $q \xrightarrow{a[\emptyset/x]} p$

(iv) $(q, w, \varepsilon) \vdash_P (p, w, x)$ if $q \xrightarrow{\tau[\emptyset/x]} p$.

Note, we see in the computation steps above that if $x = a_1 \cdots a_n$ then the symbol $a_1$ will be on top of the stack. The symbol at position $i$ from the top will be $a_i$, for $1 \leqslant i \leqslant n$.

The 'derives' relation $\vdash_P^*$ is the reflexive and transitive closure of the relation $\vdash_P$.

**Example 3.5.** For the push-down automaton of Figure 3.2 we have, e.g.,

$$(q_0, aabb, \varepsilon) \vdash_P (q_1, abb, 1) \vdash_P (q_1, bb, 11) \vdash_P (q_2, b, 1) \vdash_P (q_2, \varepsilon, \varepsilon) \vdash_P (q_3, \varepsilon, \varepsilon)$$

Thus $(q_0, aabb, \varepsilon) \vdash_P^* (q_3, \varepsilon, \varepsilon)$ and also $(q_0, aabb, \varepsilon) \vdash_P^* (q_2, b, 1)$ and $(q_1, abb, 1) \vdash_P^* (q_2, \varepsilon, \varepsilon)$.

For PDAs we have the following properties.

**Lemma 3.6.** Let $P = (Q, \Sigma, \Delta, \emptyset, \rightarrow, q_0, F)$ be a push-down automaton.

(a) For $w, w', v \in \Sigma^*$, $q, q' \in Q$, $x, x', y \in \Delta^*$ with $x \neq \varepsilon$, it holds that

$$(q, wv, xy) \vdash_P (q', w'v, x'y) \iff (q, w, x) \vdash_P (q', w', x')$$

(b) For $n \geqslant 0$, $w_i \in \Sigma^*$, $q_i \in Q$, $x_i \in \Delta^*$ with $x_i \neq \varepsilon$, for $1 \leqslant i < n$, $v \in \Sigma^*$ and $y \in \Delta^*$, it holds that

$$\begin{aligned} &(q_0, w_0v, x_0y) \vdash_P (q_1, w_1v, x_1y) \vdash_P \ldots \vdash_P (q_n, w_nv, x_ny) \\ &\qquad \iff \\ &(q_0, w_0, x_0) \vdash_P (q_1, w_1, x_1) \vdash_P \ldots \vdash_P (q_n, w_n, x_n) \end{aligned}$$

*Proof.* (a) We exploit Definition 3.4. If $(q, wv, xy) \vdash_P (q', w'v, x'y)$ with $x \neq \varepsilon$ then either we have (i) $q \xrightarrow{a[d/\bar{x}]} q'$ and $w = aw'$, $x = dx''$, $x' = \bar{x}x''$ for suitable $\bar{x}$ and $x''$ (by clause (i) of Definition 3.4), or we have (ii) $q \xrightarrow{\tau[d/\bar{x}]} q'$ and $w = w'$, $x = dx''$, $x' = \bar{x}x''$ for suitable $\bar{x}$ and $x''$, (by clause (ii) of Definition 3.4). So, either $(q, w, x) = (q, aw', dx'') \vdash_P (q', w', \bar{x}x'') = (q', w', x')$ or $(q, w, x) = (q, w, dx'') \vdash_P (q', w, \bar{x}x'') = (q', w', x')$.

Reversely, if $(q, w, x) \vdash_P (q', w', x')$, then either (i) $q \xrightarrow{a[d/\bar{x}]} q'$ and $w = aw'$, $x = dx''$, $x' = \bar{x}x''$ for suitable $\bar{x}$ and $x''$, or we have (ii) $q \xrightarrow{\tau[d/\bar{x}]} q'$ and $w = w'$, $x = dx''$, $x' = \bar{x}x''$ for suitable $\bar{x}$ and $x''$. It follows that $wv = aw'v$, $xy = dx''y$, $x'y = \bar{x}x''y$, or $wv = w'v$, $xy = dx''y$, $x'y = \bar{x}x''y$. Thus $(q, wv, xy) \vdash_P (q', w'v, x'y)$.

(b) By induction on $n$ using part (a). □

**Definition 3.7.** Let $P = (Q, \Sigma, \Delta, \emptyset, \rightarrow, q_0, F)$ be a push-down automaton. The language $\mathcal{L}(P)$, called the language *accepted* by the push-down automaton $P$, is given by

$$\{\, w \in \Sigma^* \mid \exists q \in F\, \exists x \in \Delta^* \colon (q_0, w, \varepsilon) \vdash_P^* (q, \varepsilon, x) \,\}$$

Note that we require $q$ to be a final state of $P$, i.e. $q \in F$, but $x$ can be any stack content.

**Example 3.8.** Consider once more the push-down automaton $P$ of Example 3.2. We verify $\mathcal{L}(P) = \{\, a^n b^n \mid n \geqslant 1 \,\}$ by means of a so-called invariant table.

| state $q$ | input $w$ | stack $x$ | |
|---|---|---|---|
| $q_0$ | $\varepsilon$ | $\varepsilon$ | |
| $q_1$ | $a^n$ | $1^n$ | $1 \leqslant n$ |
| $q_2$ | $a^n b^m$ | $1^{n-m}$ | $1 \leqslant m \leqslant n$ |
| $q_3$ | $a^n b^n$ | $\varepsilon$ | $1 \leqslant n$ |

The first column of the table lists the states in $Q$. The second column lists the input by which the state is reached (starting from the initial state with empty stack). The third column lists the contents of the stack after the state is reached while reading the input of the second column. The last column provides further constraints and relationships of indices involved. If a triple $(q, w, x)$ is listed, then it holds that $(q_0, w, \varepsilon) \vdash_P^* (q, \varepsilon, x)$.

For the PDA $P$ of Figure 3.2 the first row is obvious: Starting in state $q_0$ with empty stack $\varepsilon$ the PDA immediately leaves $q_0$. So, $P$ will only be in $q_0$ when no input is given, i.e., when the input equals $\varepsilon$. The second row captures that $P$ is in state $q_1$ after reading $a^n$. The first $a$ is read on the transition from $q_0$ to $q_1$, possibly more $a$'s are read on execution of the transition from $q_1$ to itself. In $q_1$ the stack contains as many symbols 1 as symbols $a$ have been read so far. Upon reading a symbol $b$, the PDA moves from state $q_1$ to state $q_2$. More $b$'s can be read, but no more than the number of symbols 1 on the stack. For each $b$ read, one symbol 1 is popped off the stack. When the stack has become empty, the transition from $q_2$ to $q_3$ becomes enabled. There no further input can be read. As the stack was empty upon leaving $q_2$ all symbols 1 must have been popped off, thus as many $b$s have been read as there were symbols $a$ before.

Now $q_3$ is the only final state of $P$. From the table we see that the input read to reach $q_3$ starting from the initial state $q_0$ with empty stack is of the form $a^n b^n$ for some $n \geqslant 1$. Thus $(q_0, w, \varepsilon) \vdash_P^* (q_3, \varepsilon, x)$ iff $w = a^n b^n$ with $n \geqslant 1$. This proves that $\mathcal{L}(P)$ is the language as claimed.

**Example 3.9.** Let us construct a push-down automaton for the non-regular language $L = \{ ww^R \mid w \in \{a, b\}^* \}$. The input alphabet is $\{a, b\}$. It is convenient to use $a$ and $b$ as stack symbols as well. So $\Delta = \{a, b\}$. In the initial state, a string is read and put on the stack. At some point (non-deterministically) the PDA will switch to the second state, where the stack will be read out again. Note that this will be in reversed order as a stack is last-in first-out. Termination takes place when the stack is empty again. The above is implemented by the push-down automaton $P$ given in Figure 3.3. In this figure we have $d \in \Delta$, so $d$ is either $a$ or $b$.

The invariant table belonging to $P$ is as follows. Recall, a triple $(q, w, x)$ is listed in the invariance table iff it holds that $(q_0, w, \varepsilon) \vdash_P^* (q, \varepsilon, x)$.

| state $q$ | input $w$ | stack $x$ | |
|---|---|---|---|
| $q_0$ | $w$ | $w^R$ | $w \in \{a, b\}^*$ |
| $q_1$ | $wv$ | $u$ | $w \in \{a, b\}^*,\ vu = w^R$ |
| $q_2$ | $wv$ | $\varepsilon$ | $w \in \{a, b\}^*,\ v = w^R$ |

In $q_0$ the string $w$ that is read, is stored on the stack in reversed order because of the last-in first-out regime of the stack. After the non-deterministic switch from state $q_0$ to state $q_1$ which does not affect the input nor the stack, in $q_1$ a symbol can only be read if it matches the symbol on top of the stack. So the extra input $v$ and the remainder of the stack $u$ together form $w^R$. Since $q_1$ is left only if the stack is empty, it must be the case in $q_2$ that $u = \varepsilon$, hence $v = w^R$. Therefore, only words of the form $ww^R$ are accepted. As any string can be read initially, including the empty string $\varepsilon$, we conclude that $P$ indeed accepts $L$.

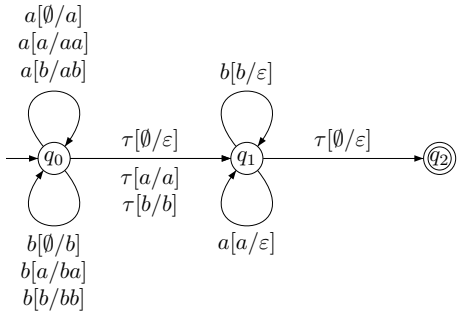



Figure 3.3: A PDA accepting $\{ ww^R \mid w \in \{a, b\}^* \}$

## 3.2 Context-free grammars

Consider the language $L = \{ a^n b^n \mid n > 0 \}$. Clearly $ab \in L$; take $n = 1$. Now, for an arbitrary element $w \in L$, the string $awb$ is also an element of $L$: if $w = a^n b^n$ for some $n > 0$, then $awb = a^{n+1}b^{n+1}$ and $n+1 > 0$. Thus, writing the symbol $S$ to represent an element of $L$, we can write

$$S \to ab \text{ and } S \to aSb \tag{3.1}$$

We may read $S \to ab$ as $S$ produces $ab$, and $S \to aSb$ as $S$ produces $aSb$. The expressions $S \to ab$ and $S \to aSb$ are called *production rules*, or more precisely production rules for $S$.

Production rules are typically used in a context yielding so-called *productions* or *derivations*. If $uSv$ is a string, say with $u, v \in \{S, a, b\}^*$, then we have the productions $uSv \Rightarrow uabv$ and $uSv \Rightarrow uaSbv$, based on the production rules $S \to ab$ and $S \to aSb$, respectively. Since the applicability of the production rules does not depend on the context comprised by the strings $u$ and $v$, but only the symbol $S$, the production scheme is referred to as context-free.

The production rules of Equation (3.1) can be applied to strings containing $S$ repeatedly, yielding sequences of zero, one or more productions or derivations. E.g., we have

$$\begin{aligned} &S \Rightarrow ab \\ &S \Rightarrow aSb \Rightarrow aabb \\ &S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb \end{aligned}$$

Such sequences are called production sequences or derivation sequences. We write, e.g., $S \Rightarrow^* ab$, $S \Rightarrow^* aabb$, $S \Rightarrow^* aaabbb$, but also $aSb \Rightarrow^* aaSbb$ and $aSb \Rightarrow^* aSb$.

**Definition 3.10** (Context-free grammar)**.** A context-free grammar (CFG) is a four-tuple $G = (V, T, R, S)$ where

1. $V$ is a non-empty finite set of variables or non-terminals,
2. $T$ is a finite set of terminals,
3. $R \subseteq V \times (V \cup T)^*$ is a finite set of production rules, and
4. $S \in V$ is the start symbol.

We use CFG as shorthand for the notion of a context-free grammar. As we shall see, the set of terminals $T$ corresponds to the input alphabet $\Sigma$ of a PDA.

If $G$ is a context-free grammar with set of production rules $R$, we suggestively write $A \to \alpha$, read $A$ produces $\alpha$, if $(A, \alpha) \in R$. Sometimes we write $A \to_G \alpha$ to stress that the production rule belongs to the grammar $G$. The production rule is called a production rule for the non-terminal $A$. If $A \to \alpha_1$, $A \to \alpha_2$, ..., $A \to \alpha_n$ are all the production rules in $R$ for $A$, we also write $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$.

**Example 3.11.** Define $G = (V, T, R, S)$ with $V = \{S\}$, $T = \{a, b\}$ and $R$ consisting of

$$S \to ab \text{ and } S \to aSb$$

Then $G$ is a context-free grammar with non-terminal $S$, terminals $a$ and $b$ and the two production rules above. The start symbol of $G$ is $S$.

**Example 3.12.** In view of the shorthand notation given above, the following describes seven production rules:

$$\begin{aligned} S &\to ABC \\ A &\to \varepsilon \mid aA \\ B &\to \varepsilon \mid Bb \\ C &\to c \mid cC \end{aligned}$$

The rules implicitly define a context-free grammar $G = (V, T, R, S)$. The following conventions apply: (i) Commonly capital letters indicate non-terminals. Thus for $G$ we have $V = \{ S, A, B, C \}$. Note that the elements of $V$ are precisely the symbols that occur at the left-hand side of the rules above. (ii) Usually, lower-case letters indicate terminals. Thus for $G$ we have $T = \{a, b, c\}$. Note that the elements of $T$ are exactly the letters that are not in $V$ and occur at the right-hand side of the rules. Note, $\varepsilon$ indicates the empty string, and hence is not in $T$, nor in $V$. (iii) Obviously, the set of production rules $R$ consists of the rules listed. (iv) A customary choice for the start symbol is $S$ which indeed is an element of the set of non-terminals $V$.

**Definition 3.13** (Production, production sequence, language of a CFG)**.** Let $G = (V, T, R, S)$ be a context-free grammar.

- Let $A \to \alpha \in R$ be a production rule of $G$. Thus $A \in V$ and $\alpha \in (V \cup T)^*$. Let $\gamma = \beta_1 A \beta_2$ be a string in which $A$ occurs. Put $\gamma' = \beta_1 \alpha \beta_2$. We say that from the string $\gamma$ the production rule $A \to \alpha$ produces the string $\gamma'$, notation $\gamma \Rightarrow_G \gamma'$.
- A production sequence or derivation is a sequence $(\gamma_i)_{i=0}^n$ such that $\gamma_{i-1} \Rightarrow_G \gamma_i$, for $1 \leqslant i \leqslant n$. Often we write

  $$\gamma_0 \Rightarrow_G \gamma_1 \Rightarrow_G \ \cdots \ \Rightarrow_G \gamma_{n-1} \Rightarrow_G \gamma_n$$

  The length of this production sequence is $n$. In case $\gamma = \gamma_0$ and $\gamma' = \gamma_n$ we also write $\gamma \Rightarrow_G^* \gamma'$.
- Let $A \in V$ be a variable of $G$. The language $\mathcal{L}_G(A)$ generated by $G$ from $A$ is given by

  $$\mathcal{L}_G(A) = \{ w \in T^* \mid A \Rightarrow_G^* w \}$$
- The language $\mathcal{L}(G)$, the language generated by the CFG $G$, consists of all strings of terminals that can be produced from the start symbol $S$, i.e.

  $$\mathcal{L}(G) = \mathcal{L}_G(S)$$

- A language $L$ is called context-free, if there exists a context-free grammar $G$ such that $L = \mathcal{L}(G)$.

Note, with respect to the grammar $G$, we have $\mathcal{L}(G) = \mathcal{L}_G(S) = \{ w \in T^* \mid S \Rightarrow_G^* w \}$. When the grammar $G$ is clear it may be dropped as a subscript.

**Example 3.14.** Consider again the grammar $G$ given in Example 3.11. As eluded to at the beginning of this section, $S \Rightarrow_G ab$ and $aaSbb \Rightarrow_G aaabbb$ are productions of $G$, but also $SS \Rightarrow_G SaSb$ is an production of $G$. Example production sequences are $S \Rightarrow_G ab$ and $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$, thus $S \Rightarrow_G^* ab$, and $S \Rightarrow_G^* aaabbb$. Also $S \Rightarrow aSb \Rightarrow aaSbb$ and $SS \Rightarrow_G SaSb \Rightarrow_G abaSb$ are production sequences of $G$. We claim $\mathcal{L}(G) = \{ a^n b^n \mid n \geqslant 1 \}$. This can be shown by proving for $L = \{ a^n b^n \mid n \geqslant 1 \}$ two set inclusions: (i) the inclusion $\mathcal{L}(G) \subseteq L$ by induction on the length of a production sequence, and (ii) the inclusion $L \subseteq \mathcal{L}(G)$ by induction on the parameter $n$.

*Proof of the claim* ($\mathcal{L}(G) \subseteq L$) We show by induction on $n$, if $S \Rightarrow_G^n w$ and $w \in T^*$ then $w \in L$. Basis: $n = 0$. Then $S = w$ while $S \notin T^*$, hence there is nothing to show. Induction step, $n + 1$: Suppose $S \Rightarrow_G^{n+1} w$. We have either $S \Rightarrow_G ab \Rightarrow_G^n w$, in case the production rule $S \to ab$ was used, or $S \Rightarrow_G aSb \Rightarrow_G^n w$, in case the production rule $S \to aSb$ was used. In the first case clearly $w \in L$. As to the second case, by Lemma 3.15c below it follows that $w = avb$ for some string $v \in T^*$ and $S \Rightarrow_G^n v$. By induction hypothesis $v \in L$, i.e. $v = a^m b^m$ for some suitable $m \geqslant 1$. Therefore $w = avb = a^{m+1} b^{m+1}$ and $w \in L$ too.

($L \subseteq \mathcal{L}(G)$) We show by induction on $n$, $S \Rightarrow_G^* a^n b^n$, for $n \geqslant 1$. Basis, $n = 1$: Clear, $S \Rightarrow_G ab$ based on the production rule $S \to ab$. Induction hypothesis, $n+1$: We have $a^n b^n \in L$. By induction hypothesis it follows that $S \Rightarrow_G^* a^n b^n$. By Lemma 3.15b we obtain $aSb \Rightarrow_G^* aa^n b^n b$. Therefore $S \Rightarrow_G aSb \Rightarrow_G^* a^{n+1} b^{n+1}$, as was to be shown.

In the example above we called to Lemma 3.15 below to split and combine production sequences. This technical lemma summarizes the context independence of the machinery introduced and is used in many situations.

**Lemma 3.15.** Let $G = (V, T, R, S)$ be a context-free grammar.

(a) Let $x, x', y, y' \in (V \cup T)^*$. If $x \Rightarrow_G^n x'$ and $y \Rightarrow_G^m y'$ then $xy \Rightarrow_G^{n+m} x'y'$.

(b) Let $k \geqslant 1$, $X_1 \ldots, X_k \in V \cup T$, $n_1, \ldots, n_k \geqslant 0$, and $x_1, \ldots, x_k \in (V \cup T)^*$. If $X_1 \Rightarrow_G^{n_1} x_1, \ldots, X_k \Rightarrow_G^{n_k} x_k$, then $X_1 \cdots X_k \Rightarrow_G^n x_1 \cdots x_k$ where $n = n_1 + \cdots + n_k$.

(c) Let $X_1 \ldots, X_k \in V \cup T$ and $x \in (V \cup T)^*$. If $X_1 \cdots X_k \Rightarrow_G^n x$ then exist $n_1, \ldots, n_k \geqslant 0$, and $x_1, \ldots, x_k \in (V \cup T)^*$ such that $n = n_1 + \cdots + n_k$, $X_1 \Rightarrow_G^{n_1} x_1, \ldots, X_k \Rightarrow_G^{n_k} x_k$, and $x = x_1 \ldots x_k$.

*Proof.* (a) By induction $n+m$, if $x \Rightarrow_G^n x'$ and $y \Rightarrow_G^m y'$ then $xy \Rightarrow_G^{n+m} x'y'$. Basis, $n+m = 0$: Trivial. We have $x = x'$, $y = y'$ and $xy \Rightarrow_G^0 xy$. Induction step, $n+m+1$: If $x \Rightarrow_G \hat{x} \Rightarrow_G^n x'$ and $y \Rightarrow_G^m y'$ we have $x = x_1 A x_2$, $\hat{x} = x_1 \bar{x} x_2$ for a production rule $A \to \bar{x}$ of $G$. By induction hypothesis $\hat{x}y \Rightarrow_G^{n+m} x'y'$. We also have $xy = x_1 A x_2 y \Rightarrow_G$

$x_1\bar{x}x_2y = \hat{x}y$. Thus $xy \Rightarrow_G^{n+m+1} x'y'$. If $y \Rightarrow_G \hat{y} \Rightarrow_G^{m} y'$, then by similar reasoning it follows that $xy \Rightarrow_G x\hat{y}$ and $x\hat{y} \Rightarrow_G^{n+m} x'y'$. From this it follows that $xy \Rightarrow_G^{n+m+1} x'y'$, as was to be shown.

(b) By induction on $k$, using part (a).

(c) By induction on $n$, if $X_1 \cdots X_k \Rightarrow_G^{n} x$ then $X_1 \Rightarrow_G^{n_1} x_1$, ..., $X_k \Rightarrow_G^{n_k} x_k$, and $x = x_1 \cdots x_k$ for suitable $n_1, \ldots, n_k$, and $x_1, \ldots, x_k$. Basis, $n = 0$: If $X_1 \cdots X_k \Rightarrow_G^{0} x$, then $X_1 \cdots X_k = x$. Choose $x_1 = X_1$, ..., $x_k = X_k$ and $n_1, \ldots, n_k = 0$. Then clearly $n_1 + \cdots + n_k = 0 = n$, $X_i \Rightarrow_G^{*} x_i$, for $1 \leqslant i \leqslant k$, and $x_1 \cdots x_k = X_1 \cdots X_k = x$. Induction step, $n+1$: Suppose

$$X_1 \cdots X_k \Rightarrow_G X_1 \cdots X_{i-1}Y_1 \cdots Y_\ell X_{i+1} \cdots X_k \Rightarrow_G^{n} x$$

for some $i$, $1 \leqslant i \leqslant k$, such that $X_i \Rightarrow_G Y_1 \cdots Y_\ell$. By induction hypothesis exist indices $n_1, \ldots, n_{i-1}, m_1, \ldots, m_\ell, n_{i+1}, \ldots, n_k$, and strings $x_1, \ldots, x_{i-1}, y_1, \ldots, y_\ell, x_{i+1}, \ldots, x_k$ such that $X_j \Rightarrow_G^{n_j} x_j$ for $1 \leqslant j < i$ or $i < j \leqslant k$, $Y_h \Rightarrow_G^{m_h} y_h$ for $1 \leqslant h \leqslant \ell$, and $n = n_1 + \cdots + n_{i-1} + m_1 + \cdots + m_\ell + n_{i+1} + \cdots + n_k$, and $x = x_1 \cdots x_{i-1}y_1 \cdots y_\ell x_{i+1} \cdots x_k$. Put $m = m_1 + \cdots + m_\ell$, $n_i = m+1$, and $x_i = y_1 \cdots y_\ell$. Then we have $n+1 = n_1 + \cdots + n_k$, and

$$X_i \Rightarrow_G Y_1 \cdots Y_\ell \Rightarrow_G^{m} y_1 \cdots y_\ell = x_i$$

Thus, by part (b), $X_i \Rightarrow_G^{n_i} x_i$. Thus $X_j \Rightarrow_G^{n_j} x_j$ for $1 \leqslant j \leqslant k$, including $i$, and $x = x_1 \cdots x_{i-1}y_1 \cdots y_\ell x_{i+1} \cdots x_k = x_1 \cdots x_k$, as was to be shown. □

**Example 3.16.** The so-called parentheses language $L_{(\,)} \subseteq \{(\,,\,)\}^*$ is the language of strings of balanced parentheses: for a string of parentheses $w = b_1 \cdots b_n \in L_{(\,)}$ it holds that for an arbitrary prefix $v \preccurlyeq w$, say $v = b_1 \cdots b_m$, $0 \leqslant m \leqslant n$, it holds that $\#_{(}(v) \geqslant \#_{)}(v)$, and for $w$ it holds that $\#_{(}(w) = \#_{)}(w)$. Thus, the $i$-th right parenthesis occurs only after the $i$-th left parenthesis, and for the left parenthesis with number $i$ there is a right parenthesis with number $i$.

We have $()(()) \in L_{(\,)}$ by inspection of the string $(_1)_1(_2(_3)_2)_3$ where we number the left and right parentheses. But $w = ())($ and $v = (()($ are not in $L_{(\,)}$ as the annotated versions $(_1)_1)_2(_2$ and $(_1(_2)_1(_3$ show; for $w$ we have that $)_2$ occurs before $(_2$, while for $v$ we have that there are 3 left parentheses but only 1 right parenthesis. Note that the empty string $\varepsilon$ meets the requirements, although it has no parenthesis at all.

A possible grammar $G$ for $L_{(\,)}$ is given by the rules

$$S \to \varepsilon,\ S \to SS,\ S \to (S)$$

With respect to $G$ we have the production sequence

$$S \Rightarrow_G SS \Rightarrow_G S(S) \Rightarrow_G S((S)) \Rightarrow_G S(()) \Rightarrow_G (S)(()) \Rightarrow_G ()(())$$

for the string $()(()) \in L_{(\,)}$, but also the production sequence

$$S \Rightarrow_G SS \Rightarrow_G (S)S \Rightarrow_G ()S \Rightarrow_G ()(S) \Rightarrow_G ()((S)) \Rightarrow_G ()(())$$

as well as

$$S \Rightarrow_G SS \Rightarrow_G (S)S \Rightarrow_G (S)(S) \Rightarrow_G ()(S) \Rightarrow_G ()((S)) \Rightarrow_G ()(())$$

There are several more derivations for ()(()) with respect to $G$.

**Definition 3.17.** Let $G = (V, T, R, S)$ be a context-free grammar. A production $\gamma \Rightarrow_G \gamma'$ is called a leftmost production if $\gamma'$ is obtained from $\gamma$ by application of a production rule of $G$ on the leftmost variable occurring in $\gamma$, i.e., $\gamma = wA\beta$, $A \to \alpha$ a rule of $G$, $\gamma' = w\alpha\beta$ for some $w \in T^*$, $A \in V$, and $\alpha, \beta \in (V \cup T)^*$. Notation, $\gamma \overset{\ell}{\Rightarrow}_G \gamma'$. A leftmost derivation of $G$ is a production sequence $(\gamma_i)_{i=0}^n$ of $G$ for which every production is leftmost. Notation, $\gamma \overset{\ell}{\Rightarrow}_G^* \gamma'$.

Similarly one can define the notion of a rightmost production and a rightmost derivation for a CFG $G$.

We often write $\gamma \overset{\ell}{\Rightarrow}^* \gamma'$ rather than $\gamma \overset{\ell}{\Rightarrow}_G^* \gamma'$, if the grammar $G$ is clear from the context. The notion of a leftmost derivation is useful to select a particular production sequence from the many that may produce a string. However, it is not generally the case that there exists precisely one leftmost derivation sequence from a string $\gamma$ to a string $\gamma'$. Although, in a leftmost derivation there is no freedom to select the variable that is used for the production, there may be freedom to select the production rule to use. Consider, for example the grammar

$$S \to AB,\ A \to aa \mid aac,\ B \to bb \mid cbb$$

which allows four complete leftmost derivations starting from $S$, viz.

$$S \overset{\ell}{\Rightarrow} AB \overset{\ell}{\Rightarrow} aaB \overset{\ell}{\Rightarrow} aabb$$
$$S \overset{\ell}{\Rightarrow} AB \overset{\ell}{\Rightarrow} aaB \overset{\ell}{\Rightarrow} aacbb$$
$$S \overset{\ell}{\Rightarrow} AB \overset{\ell}{\Rightarrow} aacB \overset{\ell}{\Rightarrow} aacbb$$
$$S \overset{\ell}{\Rightarrow} AB \overset{\ell}{\Rightarrow} aacB \overset{\ell}{\Rightarrow} aaccbb$$

Note, there are two leftmost derivations of the string *aacbb*.

In the next section we will show that if there is a production sequence from $\gamma$ to $\gamma'$ for some grammar $G$, then there is also a leftmost derivation from $\gamma$ to $\gamma'$ for $G$, i.e. if $\gamma \Rightarrow_G^* \gamma'$ then $\gamma \overset{\ell}{\Rightarrow}_G^* \gamma'$.

**Theorem 3.18.** If $L$ is a regular language, then $L$ is also context-free.

*Proof.* Let $D = (Q, \Sigma, \delta, q_0, F)$ be a deterministic automaton that accepts $L$. Define the grammar $G = (Q, \Sigma, R, q_0)$ where $R$ is given by

$$R = \{\, q \to aq' \mid \delta(q, a) = q' \,\} \cup \{\, q \to \varepsilon \mid q \in F \,\}$$

We claim that $L = \mathcal{L}(G)$.

Suppose $w \in L$ and $w = a_1a_2 \ldots a_n$. Let $(q_0, a_1a_2 \cdots a_n) \vdash_D (q_1, a_2 \cdots a_n) \vdash_D \cdots \vdash_D (q_{n-1}, a_n) \vdash_D (q_n, \varepsilon)$ with $q_n \in F$ be an accepting transition sequence of $D$ for $w$. Then we have a production sequence

$$\gamma_0 \Rightarrow_G \gamma_1 \Rightarrow_G \cdots \Rightarrow_G \gamma_{n+1}$$

of $G$ for $w$ with $\gamma_0 = q_0$, $\gamma_i = a_1 \cdots a_i q_i$ for $0 \leqslant i \leqslant n$, and $\gamma_{n+1} = a_1 \cdots a_n$. Since $\delta(q_{i-1}, a_i) = q_i$ we have $q_{i-1} \to a_i q_i$, and hence $\gamma_{i-1} = a_1 \cdots a_{i-1} q_{i-1} \Rightarrow_G qa_1 \cdots a_{i-1} a_i q_i = \gamma_i$ for $1 \leqslant i \leqslant n$ and $\gamma_n = a_1 \cdots a_n q_n \Rightarrow_G a_1 \cdots a_n = \gamma_{n+1}$. Thus $w \in \mathcal{L}(G)$ and $L \subseteq \mathcal{L}(G)$.

Suppose $w \in \mathcal{L}(G)$. Say $\gamma_0 \Rightarrow_G \gamma_1 \Rightarrow_G \cdots \Rightarrow_G \gamma_{n+1}$ for some $n \geqslant 0$ with $\gamma_0 = q_0$ and $\gamma_{n+1} = w$. Then there exist $a_1, \ldots, a_n \in \Sigma$ and $q_0, \ldots, q_n \in Q$ such that $\gamma_i = a_1 \cdots a_i q_i$, for $0 \leqslant i \leqslant n$ and $\gamma_{n+1} = w = a_1 \cdots a_n$. Moreover, $\delta(q_{i-1}, a_i) = q_i$, for $1 \leqslant i \leqslant n$, and $q_n \in F$. Therefore we have

$$(q_0, a_1a_2 \cdots a_n) \vdash_D (q_1, a_2 \cdots a_n) \vdash_D \cdot \vdash_D (q_{n-1}, a_n) \vdash_D (q_n, \varepsilon) \text{ and } q_n \in F$$

It follows that $w \in L$ and $\mathcal{L}(G) \subseteq L$. □

As we shall see later, $L_{(\,)}$ of Example 3.16 is not regular. So, the reverse of Theorem 3.18 does not hold.

## 3.3 Parse trees

In the previous section, in Example 3.16, we saw various production sequences for the same string. Basically, the particular production sequences are not very different. It is the order in which the various rules are applied that varies.

If we visualize the three derivations given in Example 3.16

$$\begin{array}{l} S \Rightarrow_G SS \Rightarrow_G S(S) \Rightarrow_G S((S)) \Rightarrow_G S(()) \Rightarrow_G (S)(()) \Rightarrow_G ()(()) \\ S \Rightarrow_G SS \Rightarrow_G (S)S \Rightarrow_G ()S \Rightarrow_G ()(S) \Rightarrow_G ()((S)) \Rightarrow_G ()(()) \\ S \Rightarrow_G SS \Rightarrow_G (S)S \Rightarrow_G (S)(S) \Rightarrow_G ()(S) \Rightarrow_G ()((S)) \Rightarrow_G ()(()) \end{array}$$

as a rooted tree, we obtain in all three cases the picture given in Figure 3.4. The end result is the same, although the tree is built up differently. In fact, the order of independent productions is abstracted away. Only the causal order remains. Note, an internal node, i.e. a node that is not a leave, together with its children corresponds to a production rule. A leaf labeled $\varepsilon$ has no siblings.

We formalize the above idea in the notion of parse tree for a context-free grammar.

**Definition 3.19.** Let $G = (V, T, R, S)$ be a context-free grammar. The collection $\mathcal{PT}_G$ of parse trees of $G$, a set of rooted node-labeled trees, is given as follows.

- A single root tree $[X]$ with root labeled $X$ is a parse tree for $G$ if $X$ is a variable or terminal of $G$.

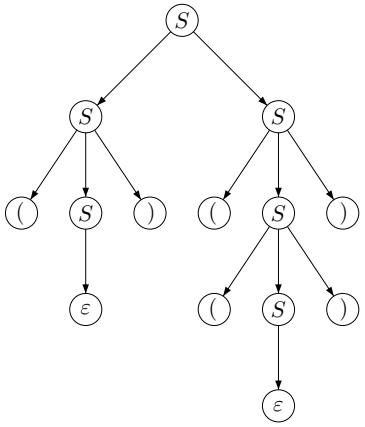



Figure 3.4: A parse tree representing three (and more) production sequences

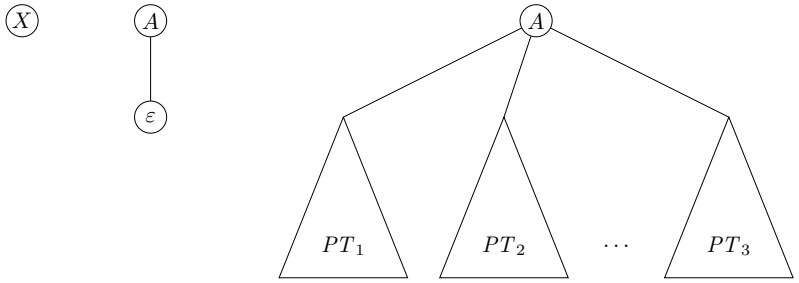



Figure 3.5: Parse tree formation

- A two-node tree $[A \to \varepsilon]$, with root labeled $A$ and leaf labeled $\varepsilon$ is a parse tree for $G$ if $A \in V$ and $A \to \varepsilon$ is a production rule of $G$.
- If $PT_1$, $PT_2$, ..., $PT_k$ are $k$ parse trees for $G$ with roots $X_1$, $X_2$, ..., $X_k$, respectively, and $A \to X_1X_2\cdots X_k$ is a production rule of $G$, then the tree $[A \to PT_1, PT_2, \ldots, PT_k]$ with root labeled $A$ and subtrees of the root $PT_1$, $PT_2$, ..., $PT_k$ is a parse tree for $G$.

The yield function $yield : \mathcal{PT}_G \to (V \cup T)^*$ with respect to $G$ is given by

- $yield([X]) = X$ for $X \in V \cup T$,
- $yield([A \to \varepsilon]) = \varepsilon$ for $A \in V$,
- $yield([A \to PT_1, PT_2, \ldots, PT_k]) = yield(PT_1) \cdot yield(PT_2) \cdot \ldots \cdot yield(PT_k)$ for $A \in V$, $PT_1, \ldots, PT_k \in \mathcal{PT}_G$.

A parse tree $PT$ for $G$ is called complete if $yield(PT) \in T^*$.

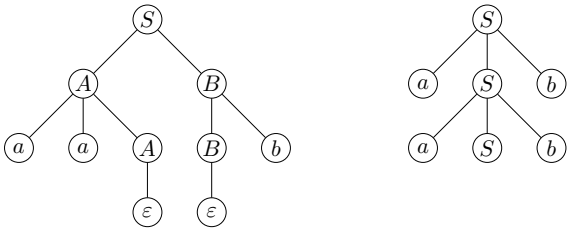



Figure 3.6: Complete parse tree yield *aab*; incomplete parse tree, yield *aaSbb*.

For a context-free grammar $G$ , we use $\mathcal{PT}_G(A)$ to denote the set of all complete parse trees of $G$ with root labeled $A$.

Next we relate derivations of a context-free grammar and its parse trees.

**Theorem 3.20.** Let $G = (V, T, R, S)$ be a context-free grammar. For $A \in V$ and $w \in T^*$, if $A \Rightarrow_G^* w$ then $w = \mathit{yield}(PT)$ for some complete parse tree $PT$ of $G$ with root $A$.

*Proof.* We prove, for $A \in V$ and $w \in T^*$,

$$A \Rightarrow_G^n w \quad \text{implies} \quad \exists PT \in \mathcal{PT}_G(A) \colon w = \mathit{yield}(PT)$$

by induction on $n$.

Basis, $n = 0$: Since $A \notin T^*$ there is nothing to prove. Induction step, $n+1$: Suppose $A \Rightarrow_G X_1 \cdots X_k \Rightarrow_G^n w$ for a production rule $A \to X_1 \cdots X_k$ of $G$. By Lemma 3.15 we can find strings $w_1, \ldots, w_k \in T^*$ such that $X_i \Rightarrow_G^* w_i$, for $1 \leqslant i \leqslant k$, and $w_1 \cdots w_k = w$. If $X_i \in T$, for $1 \leqslant i \leqslant k$, we have $X_i = w_i$ and we put $PT_i = [X_i]$. If $X_i \in V$, for $1 \leqslant i \leqslant k$, we obtain from the induction hypothesis a parse tree $PT_i$ of $G$ with root $X_i$ and yield $w_i$. Consider the parse tree $PT = [A \to PT_1, \ldots, PT_k]$ obtained from the juxaposition of the parse trees $PT_1, \ldots, PT_k$ and putting the variable $A$ on top. Since $A \to X_1 \cdots X_k \in R$ this is indeed a parse tree of $G$, it has root $A$ and is such that

$$\mathit{yield}(PT) = \mathit{yield}(PT_1) \cdots \mathit{yield}(PT_k) = w_1 \cdots w_k = w$$

This was to be shown. □

**Corollary 3.21.** For a context-free grammar $G$ with start symbol $S$ it holds that $\mathcal{L}(G) = \{\, \mathit{yield}(PT) \mid PT \in \mathcal{PT}_G(S) \,\}$.

*Proof.* Direct from the definition of $\mathcal{L}(G)$ and the theorem. □

We next show that if a string of terminals is the yield of a parse tree, then the string can be obtained by a leftmost derivation. For the proof of this result we need the following lemma.

**Lemma 3.22.** Let $G$ be a context-free grammar.

- $X \overset{\ell}{\Rightarrow}{}^*_G \alpha$ implies $wX\beta \overset{\ell}{\Rightarrow}{}^*_G w\alpha\beta$ for all $X \in (V \cup T)$, $\alpha, \beta \in (V \cup T)^*$, and $w \in T^*$.
- $w\alpha \overset{\ell}{\Rightarrow}{}^*_G w\beta$ implies $\alpha \overset{\ell}{\Rightarrow}{}^*_G \beta$ for all $w \in T^*$, and $\alpha, \beta \in (V \cup T)^*$.

*Proof.* Left as an exercise. □

We now turn to the announced result that relates parse trees and leftmost derivations. In its proof we make use of the height of a parse tree. This is the usual height of a tree, i.e. the maximal number of edges of a path from the root to a leaf.

**Theorem 3.23.** Let $G$ be a context-free grammar. Suppose $G$ has a parse tree $PT$ with root labeled $A$ and yield $w \in T^*$. Then there exists a left-most derivation $A \overset{\ell}{\Rightarrow}{}^*_G w$.

*Proof.* By induction on the height of the parse tree $PT$.

Basis, $n = 1$: Then $PT$ has root labeled $A$ and either one leaf labeled $\varepsilon$ and $w = \varepsilon$, or $k$ leaves for some $k \geqslant 1$ labeled $a_1$, ..., $a_k$ and $w = a_1 \cdots a_k$. In both cases, $A \to w$ is a production rule of $G$. Since $A \to w$ we also have $A \overset{\ell}{\Rightarrow}{}^*_G w$.

Induction step, $n + 1$: Then $PT$ has the form $[A \to PT_1, \ldots, PT_k]$, for some $k \geqslant 1$, each $PT_i$ a parse tree of height at most $n$ say with root labeled $X_i$ and yield $w_i$, $1 \leqslant i \leqslant k$, and $A \to X_1 \cdots X_k$ a production rule of $G$. Moreover, $w = w_1 \cdots w_k$.

We claim the existence of a left-most derivation $A \overset{\ell}{\Rightarrow}{}^*_G w_1 \cdots w_i X_{i+1} \cdots X_k$ for $0 \leqslant i \leqslant k$. We prove this claim by induction on $i$. Basis, i=0: Since $G$ has a production rule $A \to X_1 \cdots X_k$, we also have $A \overset{\ell}{\Rightarrow}{}^*_G X_1 \cdots X_k$, and hence we are done. Induction step, $i + 1$: By induction hypothesis for $i$ we have $A \overset{\ell}{\Rightarrow}{}^*_G w_1 \cdots w_i X_{i+1} \cdots X_k$. By induction hypothesis for $n$ we have $X_{i+1} \overset{\ell}{\Rightarrow}{}^*_G w_{i+1}$. With the help of Lemma 3.22 we obtain that

$$A \overset{\ell}{\Rightarrow}{}^*_G w_1 \cdots w_i X_{i+1} X_{i+2} \cdots X_k \overset{\ell}{\Rightarrow}{}^*_G w_1 \cdots w_i w_{i+1} X_{i+2} \cdots X_k$$

This proves the claim.

By choosing $i = k$ in the claim, we obtain $A \overset{\ell}{\Rightarrow}{}^*_G w_1 \cdots w_k$, i.e., $A \overset{\ell}{\Rightarrow}{}^*_G w$, as was to be shown. □

We summarize the results of this section as the following theorem, which states that for a context-free grammar, the set of strings obtained by arbitrary derivations, the set of strings that arise as the yield of a parse tree, and the set of strings obtained by leftmost derivations are all the same set.

**Theorem 3.24.** Let $G = (V, T, R, S)$ be a context-free grammar. Then it holds that

$$\mathcal{L}(G) = \{ \mathit{yield}(PT) \in T^* \mid PT \in \mathcal{PT}_G(S) \} = \{ w \in T^* \mid S \overset{\ell}{\Rightarrow}{}^*_G w \}$$

*Proof.* From Corollary 3.21 we obtain

$$\mathcal{L}(G) \subseteq \{\ \mathit{yield}(PT) \in T^* \mid PT \in \mathcal{PT}_G(S)\ \}$$

By Theorem 3.23 we have

$$\{\ \mathit{yield}(PT) \in T^* \mid PT \in \mathcal{PT}_G(S)\ \} \subseteq \{\ w \in T^* \mid S \stackrel{\ell}{\Rightarrow}{}^*_G\ w\ \}$$

By definition of $\mathcal{L}(G)$

$$\{\ w \in T^* \mid S \stackrel{\ell}{\Rightarrow}{}^*_G\ w\ \} \subseteq \mathcal{L}(G)$$

It follows that all set inclusions are equalities, as was to be shown. □

## 3.4 The class of context-free languages

In this section we establish that a language is generated by a context-free grammar precisely when it is accepted by a push-down automaton. We introduce the notion of acceptance by empty stack for a PDA and obtain the main result in three steps: (i) from CFG to PDA, (ii) from PDA to PDA accepting on empty stack, (iii) from PDA accepting on empty stack to CFG.

**Theorem 3.25.** Let $G = (\ V,\ T,\ R,\ S\ )$ be a context-free grammar. Then there exists a push-down automaton $P$ such that $\mathcal{L}(P) = \mathcal{L}(G)$.

*Proof.* Define $P = (\ \{q_0, q_1, q_2\},\ T,\ V \cup T,\ \emptyset, \rightarrow, q_0, \{q_2\}\ )$ with

$$\begin{array}{lll}
(1) & q_0 \xrightarrow{\tau[\emptyset/S]} q_1 & \\
(2) & q_1 \xrightarrow{a[a/\varepsilon]} q_1 & \text{for all } a \in T \text{ (matching step)} \\
(3) & q_1 \xrightarrow{\tau[\emptyset/\varepsilon]} q_2 & \\
(4) & q_1 \xrightarrow{\tau[A/\alpha]} q_1 & \text{for all } A \to \alpha \in R \text{ (production step)}
\end{array}$$

We will prove $\mathcal{L}(P) = \mathcal{L}(G)$. For this we need to claim, for all $\gamma \in (V \cup T)^*$ and $w \in T^*$,

$$\gamma \stackrel{\ell}{\Rightarrow}{}^*_G\ w \quad \text{iff} \quad (q_1, w, \gamma)\ \vdash^*_P\ (q_1, \varepsilon, \varepsilon) \tag{3.2}$$

*Proof of the claim*: ($\Rightarrow$) We show by induction on $n$, $\gamma \stackrel{\ell}{\Rightarrow}{}^n_G\ w$ implies $(q_1, w, \gamma)\ \vdash^*_P\ (q_1, \varepsilon, \varepsilon)$. Basis, $n = 0$: Then we have $\gamma = w$. By clause (2) in the definition of $P$ we have $(q_1, w, \gamma) = (q_1, w, w)\ \vdash^*_P\ (q_1, \varepsilon, \varepsilon)$. Induction step, $n+1$: Suppose $\gamma \stackrel{\ell}{\Rightarrow}_G \gamma' \stackrel{\ell}{\Rightarrow}{}^n_G\ w$. Then $\gamma = vA\beta$ for suitable $v \in T^*$, $A \in V$, and $\beta \in (V \cup T)^*$. Then we have $\gamma' = v\alpha\beta$ for some $A \to_G \alpha \in R$. Since $\gamma' \stackrel{\ell}{\Rightarrow}{}^n_G\ w$ we have that $w = vu$ and $\alpha\beta \stackrel{\ell}{\Rightarrow}{}^n_G\ u$, see Lemma 3.22. Therefore

$$\begin{array}{lll}
(q_1, w, \gamma) = (q_1, vu, vA\beta) & & \\
\quad \vdash^*_P & (q_1, u, A\beta) & \text{by clause (2)} \\
\quad \vdash_P & (q_1, u, \alpha\beta) & \text{by clause (4), } A \to_G \alpha \\
\quad \vdash^*_P & (q_1, \varepsilon, \varepsilon) & \text{by induction hypothesis, } \alpha\beta \stackrel{\ell}{\Rightarrow}{}^n_G\ u
\end{array}$$

as was to be shown.

($\Leftarrow$) By induction on $n$, if $(q_1, w, \gamma) \vdash_P^n (q_1, \varepsilon, \varepsilon)$ then $\gamma \overset{\ell}{\Rightarrow}_G^* w$. Basis, $n = 0$: Then we have $w = \varepsilon$ and $\gamma = \varepsilon$. Clearly $\varepsilon \overset{\ell}{\Rightarrow}_G^* \varepsilon$.

Induction step, $n + 1$: Suppose

$$(q_1, w, \gamma) \vdash_P (q_1, w', \gamma') \vdash_P^n (q_1, \varepsilon, \varepsilon)$$

We distinguish two cases. Case I, $\gamma = A\beta$: We then have $(q_1, w, \gamma) \vdash_P (q_1, w', \gamma')$ by clause (4) for a rule $A \rightarrow_G \alpha$. Thus $w = w'$ and $\gamma' = \alpha\beta$. By induction hypothesis $\gamma' \overset{\ell}{\Rightarrow}_G^* w'$, i.e. $\alpha\beta \overset{\ell}{\Rightarrow}_G^* w$. Thus $\gamma = A\beta \overset{\ell}{\Rightarrow}_G \alpha\beta \overset{\ell}{\Rightarrow}_G^* w$. Case II, $\gamma = a\beta$. We then have $(q_1, w, \gamma) \vdash_P (q_1, w', \gamma')$ by clause (2). Thus $w = aw'$ and $\gamma = a\gamma'$. Since $(q_1, w', \gamma') \vdash_P^n (q_1, \varepsilon, \varepsilon)$ we have $\gamma' \overset{\ell}{\Rightarrow}_G^* w'$ by induction hypothesis. Using Lemma 3.22 it follows that $a\gamma' \overset{\ell}{\Rightarrow}_G^* aw'$, i.e. $\gamma \overset{\ell}{\Rightarrow}_G^* w$.

We finally prove $\mathcal{L}(P) = \mathcal{L}(G)$. Suppose $w \in \mathcal{L}(P)$. Thus $(q_0, w, \varepsilon) \vdash_P^* (q_2, \varepsilon, \beta)$ for some string $\beta \in (V \cup T)^*$. By definition of $P$ it follows that

$$(q_0, w, \varepsilon) \vdash_P (q_1, w, S) \vdash_P^* (q_1, \varepsilon, \varepsilon) \vdash_P (q_2, \varepsilon, \varepsilon)$$

Since $(q_1, w, S) \vdash_P^* (q_1, \varepsilon, \varepsilon)$ we have by claim (3.2) $S \overset{\ell}{\Rightarrow}_G^* w$. Thus $w \in \mathcal{L}(G)$. Reversely, suppose $w \in \mathcal{L}(G)$. Thus $S \overset{\ell}{\Rightarrow}_G^* w$. Again by claim (3.2) we obtain $(q_1, w, S) \vdash_P^* (q_1, \varepsilon, \varepsilon)$. Adding the derivation steps $(q_0, w, \varepsilon) \vdash_P (q_1, w, S)$ and $(q_1, \varepsilon, \varepsilon) \vdash_P (q_2, \varepsilon, \varepsilon)$ we see $(q_0, w, \varepsilon) \vdash_P^* (q_2, \varepsilon, \varepsilon)$. Thus $w \in \mathcal{L}(P)$. □

Next we introduce the idea of a PDA accepting on empty stack.

**Definition 3.26** (Push-down automaton accepting on empty stack)**.** A push-down automaton accepting on empty stack, also called an empty-stack automaton, is a six-tuple $P = (\, Q,\, \Sigma,\, \Delta,\, D,\, \rightarrow_P,\, q_0\,)$ where

1. $Q$ is a finite set of states,
2. $\Sigma$ is a finite input alphabet,
3. $\Delta$ is a finite data alphabet or stack alphabet,
4. $D \in \Delta$ is the stack bottom symbol,
5. $\rightarrow_P \subseteq Q \times \Sigma_\tau \times \Delta \times \Delta^* \times Q$, where $\Sigma_\tau = \Sigma \cup \{\tau\}$, is a finite set of transitions or steps,
6. $q_0 \in Q$ is the initial state.

Notation and the notion of a configuration extend from a PDA as given by Definition 3.2 to a PDA accepting on empty stack. Note, a PDA accepting on empty stack has no transition if the stack is empty. The idea is that, when the stacks becomes empty the PDA blocks and accepts the string read so far. As a consequence we need to adapt the

stack in the start situation. Instead of an empty stack, a PDA accepting on empty stack starts with a stack consisting of the stack bottom symbol $D$ only. No final states need to be identified. Honoring its name, this variant of PDAs accepts on the empty stack. Therefore we have

**Definition 3.27.** Let $P = (\, Q,\, \Sigma,\, \Delta,\, D,\, \rightarrow_P,\, q_0\,)$ be a PDA accepting on empty stack. The language accepted by $P$ on empty stack, notation $\mathcal{N}(P)$, is given by

$$\{\, w \in \Sigma^* \mid \exists q \in Q\colon (q_0, w, D) \;\vdash_P^*\; (q, \varepsilon, \varepsilon)\,\}$$

A PDA accepting on empty stack may terminate in any state. Acceptance or non-acceptance is decided by the stack being empty or not.

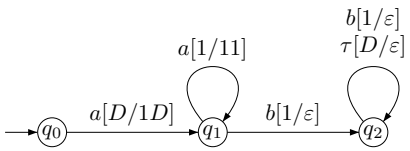



Figure 3.7: Example push-down automaton accepting on empty stack

**Example 3.28.** Figure 3.7 displays a push-down automaton $P$ accepting on empty stack. We have $P = (Q, \Sigma, \Delta, D, \rightarrow_P, q_0)$ with $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $\Delta = \{1, D\}$ now including the special symbol $D$ that is the stack bottom, and a transition relation given by

$$q_0 \xrightarrow{a[D/1D]}_P q_1, \quad q_1 \xrightarrow{a[1/11]}_P q_1, \quad q_1 \xrightarrow{b[1/\varepsilon]}_P q_2, \quad q_2 \xrightarrow{b[1/\varepsilon]}_P q_2, \quad q_2 \xrightarrow{\tau[D/\varepsilon]}_P q_2$$

Compared to the standard PDA given in Figure 3.2 there is no final state $q_3$. In the transitions of Figure 3.7 there is no mentioning of the empty stack using $\emptyset$. Testing for empty stack can now be done by testing for the new stack symbol $D$ on top (or rather bottom) of the stack.

**Theorem 3.29.** For every push-down automaton $P$, accepting on final state, exists a push-down automaton $P'$ accepting on empty stack such that $\mathcal{L}(P) = \mathcal{N}(P')$.

*Proof.* Suppose $P = (Q,\, \Sigma,\, \Delta,\, \emptyset,\, \rightarrow_P,\, q_0, F)$. Define $P'$ by

$$P' = (Q \cup \{q_{end}\},\, \Sigma,\, \Delta \cup \{D\},\, D,\, \rightarrow_{P'},\, q_0\,)$$

with a new state $q_{end}$, a new stack symbol $D$, and with a transition relation $\rightarrow_{P'}$ given

by

$$\begin{aligned}
q &\xrightarrow{a[d/x]}_{P'} q' && \text{if } q \xrightarrow{a[d/x]}_{P} q' \\
q &\xrightarrow{a[D/xD]}_{P'} q' && \text{if } q \xrightarrow{a[\emptyset/x]}_{P} q' \\
q &\xrightarrow{\tau[d/x]}_{P'} q' && \text{if } q \xrightarrow{\tau[d/x]}_{P} q' \\
q &\xrightarrow{\tau[D/xD]}_{P'} q' && \text{if } q \xrightarrow{\tau[\emptyset/x]}_{P} q' \\
q &\xrightarrow{\tau[d/d]}_{P'} q_{end} && \text{for } q \in F \text{ and } d \in \Delta \cup \{D\} \\
q_{end} &\xrightarrow{\tau[d/\varepsilon]}_{P'} q_{end} && \text{for } d \in \Delta \\
q_{end} &\xrightarrow{\tau[D/\varepsilon]}_{P'} q_{end}
\end{aligned}$$

It holds that $(q, w, x) \vdash_P (q', w', x')$ iff $(q, w, xD) \vdash_{P'} (q', w', x'D)$ for $w \in \Sigma^*$, $q, q' \in Q$, and $x, x' \in \Delta^*$. From this it follows that $(q_0, w, \varepsilon) \vdash_P^* (q, \varepsilon, x)$ for some $q \in F$ iff $(q_0, w, D) \vdash_{P'}^* (q, \varepsilon, xD)$ for some $q \in F$ iff $(q_0, w, D) \vdash_{P'}^* (q_{end}, \varepsilon, \varepsilon)$, since $q_{end}$ can only be reached in $P'$ via a final state of $P$. We conclude $w \in \mathcal{L}(P)$ iff $w \in \mathcal{N}(P')$, and $\mathcal{L}(P) = \mathcal{N}(P')$ as was to be shown. □

The above theorem connects the two types of push-down automata in one direction. The next result connects a push-down automaton accepting on empty stack with a context-free grammar.

**Theorem 3.30.** Let $P = (\, Q, \Sigma, \Delta, \rightarrow_P, q_0, D\,)$ be a push-down automaton accepting on empty stack. There exists a context-free grammar $G$ such that $\mathcal{L}(G) = \mathcal{N}(P)$.

*Proof.* We first define the grammar $G$. Its terminal alphabet $T$ is the input alphabet $\Sigma$ of $P$. The variable alphabet $V$ contains a start symbol $S$ and variables of the form $[q\,d\,q']$ where $q$ and $q'$ are states in $Q$ and $d$ is a stack symbol in $\Delta$. Thus

$$V = \{S\} \cup \{\, [q\,d\,q'] \mid q, q' \in Q,\, d \in \Delta \,\}$$

The rules of $G$ are of three types

- $S \to [q_0\,D\,q']$ for all $q' \in Q$
- $[q\,d\,q''] \to a[p_0\,d_1\,p_1][p_1\,d_2\,p_2]\cdots[p_{k-1}\,d_k\,p_k]$ if $q \xrightarrow{a[d/d_1\cdots d_k]}_P q'$ with $p_0, \ldots, p_k \in Q$ such that $p_0 = q'$ and $p_k = q''$
- $[q\,d\,q''] \to [p_0\,d_1\,p_1][p_1\,d_2\,p_2]\cdots[p_{k-1}\,d_k\,p_k]$ if $q \xrightarrow{\tau[d/d_1\cdots d_k]}_P q'$ with $p_0, \ldots, p_k \in Q$ such that $p_0 = q'$ and $p_k = q''$

In order to show the equality of $\mathcal{L}(G)$ and $\mathcal{N}(P)$ we prove

$$[q\,d\,q'] \Rightarrow_G^* w \quad \text{iff} \quad (q, w, d) \vdash_P^* (q', \varepsilon, \varepsilon) \tag{3.3}$$

($\Rightarrow$) We prove, by induction on $n$, that $[q\,d\,q'] \Rightarrow_G^n w$ implies $(q, w, d) \vdash_P^* (q', \varepsilon, \varepsilon)$.

Basis, $n = 1$: If $[q\,d\,q'] \Rightarrow_G w$ then $[q\,d\,q'] \to w$ is a rule of $G$. So, $w = a$ for some $a \in \Sigma$ and $q \xrightarrow{a[d/\varepsilon]} q'$ or $w = \varepsilon$ and $q \xrightarrow{\tau[d/\varepsilon]} q'$. Thus $(q, a, d) \vdash_P (q', \varepsilon, \varepsilon)$ or $(q, \varepsilon, d) \vdash_P (q', \varepsilon, \varepsilon)$ and hence $(q, a, d) \vdash_P^* (q', \varepsilon, \varepsilon)$ or $(q, \varepsilon, d) \vdash_P^* (q', \varepsilon, \varepsilon)$.

Induction step, $n + 1$: We have either have

$$[q\,d\,q'] \Rightarrow_G a[p_0\,d_1\,p_1]\cdots[p_{k-1}\,d_k\,p_k] \Rightarrow_G^n w$$

for some rule $[q\,d\,q'] \to a[p_0\,d_1\,p_1]\cdots[p_{k-1}\,d_k\,p_k]$ of $G$, or

$$[q\,d\,q'] \Rightarrow_G [p_0\,d_1\,p_1]\cdots[p_{k-1}\,d_k\,p_k] \Rightarrow_G^n w$$

for some rule $[q\,d\,q'] \to [p_0\,d_1\,p_1]\cdots[p_{k-1}\,d_k\,p_k]$ of $G$. We consider the first case, leaving the second case to the reader.

By Lemma 3.15 we have $w = aw_1 \ldots w_k$ where $[p_{i-1}\,d_i\,p_i] \Rightarrow_G^* w_i$, $1 \leqslant i \leqslant k$. Thus, by induction hypothesis, $(p_{i-1}, w_i, d_i) \vdash_P^* (p_i, \varepsilon, \varepsilon)$. By Lemma 3.6 we obtain

$$(p_{i-1}, w_i w_{i+1}\cdots w_k, d_i d_{i+1}\cdots d_k) \vdash_P^* (p_i, w_{i+1}\cdots w_k, d_{i+1}\cdots d_k)$$

for $1 \leqslant i \leqslant k$. By putting these $k$ derivations together we obtain that

$$(p_0, w_1\cdots w_k, d_1\cdots d_k) \vdash_P^* (p_k, \varepsilon, \varepsilon)$$

Since $(q, aw, d) = (q, aw_1\cdots w_k, d) \vdash_P (p_0, w_1\cdots w_k, d_1\cdots d_k)$ we conclude $(q, aw, d) \vdash_P^* (q', \varepsilon, \varepsilon)$.

($\Leftarrow$) We prove, by induction on $n$, $(q, w, d) \vdash_P^n (q', \varepsilon, \varepsilon)$ implies $[q\,d\,q'] \Rightarrow_G^* w$. Basis, $n = 1$: If $(q, w, d) \vdash_P (q', \varepsilon, \varepsilon)$, we have $w = a$ for some transition $q \xrightarrow{a[d/\varepsilon]}_P q'$ or $w = \varepsilon$ for some transition $q \xrightarrow{\tau[d/\varepsilon]}_P q'$. Thus $G$ has the production rule $[q\,d\,q'] \to a$ or $[q\,d\,q'] \to \varepsilon$ and $[q\,d\,q'] \Rightarrow_G^* w$. Induction step, $n + 1$: Suppose

$$(q, w, d) \vdash_P (r, v, d_1\cdots d_k) \vdash_P^n (q', \varepsilon, \varepsilon)$$

based on the transition $q \xrightarrow{a[d/d_1\cdots d_k]}_P r$. As all of $d_1, \ldots, d_k$ are successively popped of the stack in this derivation, we can find states $r_0, \ldots r_k \in Q$, strings $v_0, \ldots, v_k \in \Sigma^*$ such that $r_0 = r$, $v_0 = v$, and

$$\begin{aligned}(r_0, v_0, d_1\cdots d_k) &\vdash_P^* (r_1, v_1, d_2\cdots d_k) \vdash_P^* \cdots \\ &\vdash_P^* (r_{k-1}, v_{k-1}, d_k) \vdash_P^* (r_k, v_k, \varepsilon) \vdash_P^* (q', \varepsilon, \varepsilon)\end{aligned}$$

Note, since $P$ terminates on empty stack, $r_k = q'$, $v_k = \varepsilon$. By Lemma 3.6 we obtain

$$\begin{gathered}(r_0, v_0\backslash v_1, d_1) \vdash_P^* (r_1, \varepsilon, \varepsilon), \quad (r_1, v_1\backslash v_2, d_2) \vdash_P^* (r_2, \varepsilon, \varepsilon), \\ \ldots\ , \quad (r_{k-1}, v_{k-1}\backslash v_k, d_k) \vdash_P^* (r_k, \varepsilon, \varepsilon)\end{gathered}$$

If $x \succcurlyeq y$ then $x\backslash y$ is a prefix of $x$ such that $x = (y\backslash x)y$. Since the length of these derivations does not exceed $n$, it follows from the induction hypothesis that

$$[r_0\,d_1\,r_1] \Rightarrow_G^* v_0\backslash v_1, \quad [r_1\,d_2\,r_2] \Rightarrow_G^* v_1\backslash v_2 \quad \ldots\ , \quad [r_{k-1}\,d_k\,r_k] \Rightarrow_G^* v_{k-1}\backslash v_k$$

Hence, by Lemma 3.6,

$$[r_0\, d_1\, r_1][r_1\, d_2\, r_2]\cdots[r_{k-1}\, d_k\, r_k] \;\Rightarrow_G^*\; (v_0\backslash v_1)(v_1\backslash v_2)\cdots(v_{k-1}\backslash v_k)$$

Since $(v_0\backslash v_1)(v_1\backslash v_2)\cdots(v_{k-1}\backslash v_k) = v_0\backslash v_k = v\backslash\varepsilon = v$ we conclude

$$[q\, d\, q'] \;\Rightarrow_G^*\; a[r_0\, d_1\, r_1][r_1\, d_2\, r_2]\cdots[r_{k-1}\, d_k\, r_k] \;\Rightarrow_G^*\; av = w$$

The case where

$$(q, w, d) \;\vdash_P\; (r, w, d_1\cdots d_k) \;\vdash_P^n\; (q', \varepsilon, \varepsilon)$$

based on the transition $q \xrightarrow{\tau[d/d_1\cdots d_k]}_P r$ is left to the reader. This proves the claim.

With the property of Equation (3.3) in place, we finish the proof as follows: $w \in \mathcal{L}(G)$ iff $S \;\Rightarrow_G^*\; w$ iff $[q_0\, D\, q'] \;\Rightarrow_G^*\; w$ for some $q' \in Q$ iff $(q_0, w, D) \;\vdash_P\; (q', \varepsilon, \varepsilon)$ for some $q' \in Q$ iff $w \in \mathcal{N}(P)$. □

Combination of the three theorems above yields that context-free grammars and push-down automata characterize the same languages.

**Corollary 3.31.** Let $L$ be a language. The following three statements are equivalent.

(*i*) $L = \mathcal{L}(P)$ for a push-down automaton $P$ accepting on final state;

(*ii*) $L = \mathcal{N}(P')$ for a push-down automaton $P'$ accepting on empty stack;

(*iii*) $L = \mathcal{L}(G)$ for a context-free grammar $G$.

*Proof.* [(*i*) ⇒ (*ii*)] This is Theorem 3.29. [(*ii*) ⇒ (*iii*)] This is Theorem 3.30. [(*iii*) ⇒ (*i*)] This is Theorem 3.25. □

## 3.5 Properties of the class of context-free languages

In this section we establish a number of positive and negative results for the class of context-free languages. We start off with closure properties that hold for regular languages and show that these hold for context-free languages too.

We will make use of the following observation: Suppose $G_1$ and $G_2$ are two context-free grammars for two languages $L_1$ and $L_2$, respectively. Say $G_i = (\, V_i,\, \Sigma,\, R_i,\, S_i\,)$ for $1 \leqslant i \leqslant 2$. Then we can assume that $G_1$ and $G_2$ have different variables, i.e. $V_1 \cap V_2 = \emptyset$, applying a renaming of variables if needed.

**Theorem 3.32.** Let $L_1$, $L_2$ and $L$ be context-free languages over an alphabet $\Sigma$.

(a) The language $L_1 \cup L_2$, the union of $L_1$ and $L_2$, is also a context-free language.

(b) The language $L_1 \cdot L_2 = \{\, w_1 w_2 \mid w_1 \in L_1,\; w_2 \in L_2 \,\}$, the concatenation of $L_1$ and $L_2$, is also a context-free language.

(c) The language $L^* = \{\, w_1 \cdots w_n \mid n \geqslant 0,\, w_1, \ldots, w_n \in L \,\}$, the Kleene closure of $L$, is also a context-free language.

*Proof.* (a) Suppose $L_i = \mathcal{L}(G_i)$, $G_i = (V_i, \Sigma, R_i, S_i)$ for $1 \leqslant i \leqslant 2$. Consider the context-free grammar $G' = (V', \Sigma, R', S')$ where

$$V' = \{S'\} \cup V_1 \cup V_2 \quad \text{and} \quad R' = \{ S' \to S_1,\, S' \to S_2 \} \cup R_1 \cup R_2$$

for a fresh variable $S'$ not in $V_1$ or $V_2$. We claim $\mathcal{L}(G') = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$. If $w \in \mathcal{L}(G')$, then $S' \Rightarrow_{G'}^* w$. By definition of $G'$, either it holds that

$$S' \Rightarrow_{G'} S_1 \Rightarrow_{G'}^* w \quad \text{or} \quad S' \Rightarrow_{G'} S_2 \Rightarrow_{G'}^* w$$

In both cases, the derivation $S_i \Rightarrow_{G'}^* w$ of $G'$ is also a derivation $S_i \Rightarrow_{G_i}^* w$ of $G_i$, since $G_1$ and $G_2$ have disjoint sets of variables. Thus $w \in \mathcal{L}(G_1)$ or $w \in \mathcal{L}(G_2)$ and hence $w \in \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$. We conclude $\mathcal{L}(G') \subseteq \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$. If $w \in \mathcal{L}(G_1)$, then $S_1 \Rightarrow_{G_1}^* w$. By definition of $G'$, we then also have $S' \Rightarrow_{G'} S_1 \Rightarrow_{G'}^* w$. Therefore $w \in \mathcal{L}(G')$. It follows that $\mathcal{L}(G_1) \subseteq \mathcal{L}(G')$. Likewise we can derive $\mathcal{L}(G_2) \subseteq \mathcal{L}(G')$. We conclude $\mathcal{L}(G_1) \cup \mathcal{L}(G_2) \subseteq \mathcal{L}(G')$. Hence $\mathcal{L}(G_1) \cup \mathcal{L}(G_2) = \mathcal{L}(G')$.

(b) Consider the context-free grammar $G' = (V', \Sigma, R', S')$ where

$$V' = \{S'\} \cup V_1 \cup V_2 \quad \text{and} \quad R' = \{ S' \to S_1 S_2 \} \cup R_1 \cup R_2$$

for a fresh variable $S'$ not in $V_1$ or $V_2$. We claim $\mathcal{L}(G') = \mathcal{L}(G_1) \cdot \mathcal{L}(G_2)$. If $w \in \mathcal{L}(G')$, then $S' \Rightarrow_{G'}^* w$. By definition of $G'$, it holds that

$$S' \Rightarrow_{G'} S_1 S_2 \Rightarrow_{G'}^* w$$

By Lemma 3.15 we can find two words $w_1, w_2 \in \Sigma^*$ such that

$$S_1 \Rightarrow_{G'}^* w_1, \quad S_2 \Rightarrow_{G'}^* w_2, \quad \text{and} \quad w_1 w_2 = w$$

Since $V_1 \cap V_2 = \emptyset$ we then also have $S_1 \Rightarrow_{G_1}^* w_1$ and $S_2 \Rightarrow_{G_2}^* w_2$. Thus $w_1 \in \mathcal{L}(G_1)$ and $w_2 \in \mathcal{L}(G_2)$. Hence $w = w_1 w_2 \in \mathcal{L}(G_1) \cdot \mathcal{L}(G_2)$ and $\mathcal{L}(G') \subseteq \mathcal{L}(G_1) \cdot \mathcal{L}(G_2)$. If $w \in \mathcal{L}(G_1) \cdot \mathcal{L}(G_2)$, we can write $w = w_1 w_2$ for some $w_1 \in \mathcal{L}(G_1)$ and $w_2 \in \mathcal{L}(G_2)$. Thus $S_1 \Rightarrow_{G_1}^* w_1$ and $S_2 \Rightarrow_{G_2}^* w_2$ By definition of $G'$ we then also have $S_1 \Rightarrow_{G'}^* w_1$ and $S_2 \Rightarrow_{G'}^* w_2$. Therefore, by Lemma 3.15, it follows that $S_1 S_2 \Rightarrow_{G'}^* w_1 w_2$. Hence,

$$S \Rightarrow_{G'} S_1 S_2 \Rightarrow_{G'}^* w_1 w_2 = w$$

and thus $w \in \mathcal{L}(G')$. We conclude that $\mathcal{L}(G_1) \cdot \mathcal{L}(G') \subseteq \mathcal{L}(G')$.

(c) Consider the context-free grammar $G' = (V', \Sigma, R', S')$ where

$$V' = \{S'\} \cup V \quad \text{and} \quad R' = \{ S' \to \varepsilon,\, S' \to S S' \} \cup R$$

for a fresh variable $S'$ not in $V$. We claim $\mathcal{L}(G') = \mathcal{L}(G)^*$. If $w \in \mathcal{L}(G')$, then $S' \Rightarrow_{G'}^* w$. By Theorem 3.24 we can assume $S' \overset{\ell}{\Rightarrow}_{G'}^* w$. By definition of $G'$, assuming $k+1$ production sequences for $S'$, we then have

$$\begin{aligned} S' &\Rightarrow_{G'} S S' \overset{\ell}{\Rightarrow}_G^* \\ &\quad w_1 S' \Rightarrow_{G'} w_1 S S' \overset{\ell}{\Rightarrow}_G^* \cdots \overset{\ell}{\Rightarrow}_G^* \\ &\qquad w_1 \cdots w_{k-1} S' \Rightarrow_{G'} w_1 \cdots w_{k-1} S S' \overset{\ell}{\Rightarrow}_G^* \\ &\qquad\quad w_1 \cdots w_{k-1} w_k S' \Rightarrow_{G'} w_1 \cdots w_{k-1} w_k = w \end{aligned}$$

for suitable strings $w_1, \ldots, w_k \in \Sigma^*$. By definition of $G'$ and Lemma 3.15, we then also have $S \Rightarrow_G^* w_i$ for $1 \leqslant i \leqslant k$. Thus $w_1, \ldots, w_k \in \mathcal{L}(G)$. Hence $w = w_1 \cdots w_k \in \mathcal{L}(G)^*$ and $\mathcal{L}(G') \subseteq \mathcal{L}(G)^*$. If $w \in \mathcal{L}(G)^*$, then $w = w_1 \cdots w_k$ for suitable $k \geqslant 0$ and $w_1, \ldots, w_k \in \mathcal{L}(G)$. We have $S \Rightarrow_G^* w_i$ for $1 \leqslant i \leqslant k$. Therefore, by Lemma 3.15, we have

$$w_1 \cdots w_{i-1} S' \Rightarrow_{G'} w_1 \cdots w_{i-1} S S' \Rightarrow_{G'}^* w_1 \cdots w_{i-1} w_i S'$$

for $1 \leqslant i \leqslant k$. Combining these derivations with $w_1 \cdots w_k S' \Rightarrow_{G'} w_1 \cdots w_k$ and using $w = w_1 \cdots w_k$ this yields $S' \Rightarrow_{G'}^* w$. Thus $w \in \mathcal{L}(G')$ and hence $\mathcal{L}(G)^* \subseteq \mathcal{L}(G')$. □

At the end of the section, see Theorem 3.40, we will show that the intersection of two context-free languages need not to be context-free again. So, the class of context-free languages is not closed under intersection. However, a restricted closure property for intersection does hold.

**Theorem 3.33.** The intersection $L_1 \cap L_2$ of a context-free language $L_1$ and a regular language $L_2$ is a context-free language.

*Proof.* Let $P = (Q_1, \Sigma, \Delta, \emptyset, \rightarrow_P, q_0^1, F_1)$ be a push-down automaton accepting $L_1$ and let $D = (Q_2, \Sigma, \delta, q_0^2, F_2)$ be a deterministic finite automaton accepting $L_2$. Note automaton $D$ has no $\tau$-transitions. We construct a push-down automaton

$$P' = (Q_1 \times Q_2, \Sigma, \Delta, \emptyset, \rightarrow_{P'}, (q_0^1, q_0^2), F_1 \times F_2)$$

where the transition relation $\rightarrow_{P'}$ is defined by

$$\begin{aligned}
(q_1, q_2) &\xrightarrow{a[d/x]}_{P'} (q_1', q_2') \quad \text{iff} \quad q_1 \xrightarrow{a[d/x]}_P q_1' \quad \text{and} \quad \delta(q_2, a) = q_2' \\
(q_1, q_2) &\xrightarrow{\tau[d/x]}_{P'} (q_1', q_2) \quad \text{iff} \quad q_1 \xrightarrow{\tau[d/x]}_P q_1'
\end{aligned}$$

for $q_1, q_1' \in Q_1$, $q_2, q_2' \in Q_2$, $a \in \Sigma$, $d \in \Delta_\emptyset$, and $x \in \Delta^*$. We will show $\mathcal{L}(P') = \mathcal{L}(P) \cap \mathcal{L}(D)$. We first claim

$$\begin{aligned}
((q_1, q_2), w, z) \vdash_{P'}^* ((\bar{q}_1, \bar{q}_2), \varepsilon, z') \quad &\text{iff} \\
(q_1, w, z) \vdash_P^* (\bar{q}_1, \varepsilon, z') \text{ and } (q_2, w) &\vdash_D^* (\bar{q}_2, \varepsilon)
\end{aligned} \tag{3.4}$$

for states $q_1, \bar{q}_1 \in Q_1$, $q_2, \bar{q}_2 \in Q_2$, $w \in \Sigma^*$ and $z, z' \in \Delta^*$.

By induction on $n$, if $((q_1, q_2), w, z) \vdash_{P'}^n ((\bar{q}_1, \bar{q}_2), \varepsilon, z')$, then $(q_1, w, z) \vdash_P^* (\bar{q}_1, \varepsilon, z')$ and $(q_2, w) \vdash_D^* (\bar{q}_2, \varepsilon)$. Basis, $n = 0$: Then we have $w = \varepsilon$, $q_1 = \bar{q}_1$, $q_2 = \bar{q}_2$, and $z = z'$. Therefore $(q_1, w, z) \vdash_{P'}^* (\bar{q}_1, \varepsilon, z')$ and $(q_2, w) \vdash_D^* (\bar{q}_2, \varepsilon)$. Induction step, $n + 1$: Suppose

$$((q_1, q_2), w, z) \vdash_{P'} ((\tilde{q}_1, \tilde{q}_2), \tilde{w}, \tilde{z}) \vdash_{P'}^n ((\bar{q}_1, \bar{q}_2), \varepsilon, z')$$

Then either

(i) $w = a\tilde{w}$, $z = dy$, $\tilde{z} = xy$ and $(q_1, q_2) \xrightarrow{a[d/x]}_{P'} (\tilde{q}_1, \tilde{q}_2)$ because $q_1 \xrightarrow{a[d/x]}_P \tilde{q}_1$ and $\delta(q_2, a) = \tilde{q}_2$, or

(ii) $w = \tilde{w}$, $z = dy$, $\tilde{z} = xy$, $q_2 = \tilde{q}_2$, and $(q_1, q_2) \xrightarrow{\tau[d/x]}_{P'} (\tilde{q}_1, q_2)$ because $q_1 \xrightarrow{\tau[d/x]}_{P} \tilde{q}_1$

Thus either $w = a\tilde{w}$, $(q_1, w, z) \vdash_P (\tilde{q}_1, \tilde{w}, \tilde{z})$ and $q_2 \xrightarrow{a}_D \tilde{q}_2$, or $w = \tilde{w}$, $q_2 = \tilde{q}_2$, and $(q_1, w, z) \vdash_P (\tilde{q}_1, \tilde{w}, \tilde{z})$. By induction hypothesis, $(\tilde{q}^1, \tilde{w}, \tilde{z}) \vdash^*_{P'} (\bar{q}_1, \varepsilon, z')$ and $(\tilde{q}_2, \tilde{w}) \vdash^*_D (\bar{q}_2, \varepsilon)$. Therefore, in both cases $(q_1, w, z) \vdash^*_{P'} (\bar{q}_1, \varepsilon, z')$ and $(q_2, w) \vdash^*_D (\bar{q}_2, \varepsilon)$.

By induction on $n$, if $(q_1, w, z) \vdash^n_{P'} (\bar{q}_1, \varepsilon, z')$ and $(q_2, w) \vdash^*_D (\bar{q}_2, \varepsilon)$, then it holds that $((q_1, q_2), w, z) \vdash^*_{P'} ((\bar{q}_1, \bar{q}_2), \varepsilon, z')$. Basis, $n = 0$: Then we have $w = \varepsilon$, $q_1 = \bar{q}_1$, $z = z'$, and also $q_2 = \bar{q}_2$, since $D$ has no $\varepsilon$-transitions. Thus $((q_1, q_2), w, z) \vdash^*_{P'} ((\bar{q}_1, \bar{q}_2), \varepsilon, z')$.

Induction step, $n + 1$: Suppose $(q_1, w, z) \vdash_{P'} (\tilde{q}_1, \tilde{w}, \tilde{z}) \vdash^n_{P'} (\bar{q}_1, \varepsilon, z')$. We distinguish two cases. Case I: $w = a\tilde{w}$, $z = dy$, $q_1 \xrightarrow{a[d/x]}_P \tilde{q}_1$, and $\tilde{z} = xy$. Since $w = a\tilde{w}$ and $D$ is deterministic, we have $(q_2, w) \vdash_D (\tilde{q}_2, \tilde{w}) \vdash^*_D (\bar{q}_2, \varepsilon)$ for a unique state $\tilde{q}_2 \in Q_2$. Therefore, by construction of $P'$, $(q_1, q_2) \xrightarrow{a[d/x]}_{P'} (\tilde{q}_1, \tilde{q}_2)$. From $(\tilde{q}_1, \tilde{w}, \tilde{z}) \vdash^n_{P'} (\bar{q}_1, \varepsilon, z')$ and $(\tilde{q}_2, \tilde{w}) \vdash^*_D (\bar{q}_2, \varepsilon)$ and the induction hypothesis it follows that $((\tilde{q}_1, \tilde{q}_2), \tilde{w}, \tilde{z}) \vdash^*_{P'} ((\bar{q}_1, \bar{q}_2), \varepsilon, z')$. Therefore $((q_1, q_2), w, z) \vdash^*_{P'} ((\bar{q}_1, \bar{q}_2), \varepsilon, z')$ in this case. Case II: $w = \tilde{w}$, $z = dy$, $q_1 \xrightarrow{\tau[d/x]}_P \tilde{q}_1$, and $\tilde{z} = xy$. By construction of $P'$ we have $(q_1, q_2) \xrightarrow{\tau[d/x]}_{P'} (\tilde{q}_1, q_2)$. Thus, $((q_1, q_2), w, z) \vdash_{P'} ((\tilde{q}_1, q_2), w, \tilde{z})$. From $(\tilde{q}_1, w, \tilde{z}) \vdash^n_{P'} (\bar{q}_1, \varepsilon, z')$, $(q_2, w) \vdash^*_D (\bar{q}_2, \varepsilon)$, and the induction hypothesis it follows that $((\tilde{q}_1, q_2), w, \tilde{z}) \vdash^*_{P'} ((\bar{q}_1, \bar{q}_2), \varepsilon, z')$. Therefore $((q_1, q_2), w, z) \vdash^*_{P'} ((\bar{q}_1, \bar{q}_2), \varepsilon, z')$ also in this case, which concludes the induction step.

To show $\mathcal{L}(P') = \mathcal{L}(P) \cap \mathcal{L}(D)$ we reason as follows: Suppose $w \in L_1 \cap L_2$. Thus, $w \in L_1$, hence $(q^1_0, w, \varepsilon) \vdash^*_P (\bar{q}_1, \varepsilon, z)$ for some $\bar{q}_1 \in F_1$ and $z \in \Delta^*$, and $w \in L_1$, hence $(q^2_0, w) \vdash^*_D (\bar{q}_2, \varepsilon)$ for some $\bar{q}_2 \in F_2$. By the claim (3.4) it follows that $((q^1_0, q^2_0), w, \varepsilon) \vdash^*_{P'} ((\bar{q}_1, \bar{q}_2), \varepsilon, z)$ while $(\bar{q}_1, \bar{q}_2) \in F_1 \times F_2$. Therefore $w \in \mathcal{L}(P')$.

Suppose $w \in \mathcal{L}(P')$. Then $((q^1_0, q^2_0), w, \varepsilon) \vdash^*_{P'} ((\bar{q}_1, \bar{q}_2), \varepsilon, z)$ for some states $\bar{q}_1 \in F_1$, $\bar{q}_2 \in F_2$ and $z \in \Delta^*$. By the claim (3.4) it follows that $(q^1_0, w, \varepsilon) \vdash^*_P (\bar{q}_1, \varepsilon, z)$ and $(q^2_0, w) \vdash^*_D (\bar{q}_2, \varepsilon)$. Thus $w \in \mathcal{L}(P)$ and $w \in \mathcal{L}(D)$, hence $w \in \mathcal{L}_1 \cap \mathcal{L}_2$. □

**Example 3.34.** Consider the language

$$L = \{\, w \in \{a, b\}^* \mid \#_a(w) = \#_b(w),\ w \text{ has no substring } abba \,\}$$

It holds that $L$ is a context-free language. We will use Theorem 3.33 to verify this.

Choose $L_1 = \{\, w \in \{a, b\}^* \mid \#_a(w) = \#_b(w) \,\}$ as context-free language and $L_2 = \{\, w \in \{a, b\}^* \mid w \text{ has no substring } abba \,\}$ as regular language. $L_2$ is regular as it is the complement of the language $\{\, w \in \{a, b\}^* \mid w \text{ has a substring } abba \,\}$, i.e. the complement of the language of the regular expression $(a + b)^*abba(a + b)^*$. Clearly, $L = L_1 \cap L_2$, hence $L$ is context-free being the intersection of a context-free language and a regular language.

We next consider a counterpart ot the Pumping Lemma for regular languages, Theorem 2.27. Also Theorem 3.35 can be used to prove negative results, regarding a language not being context-free.

**Theorem 3.35** (Pumping Lemma for context-free languages)**.** Let $L$ be a context-free language over an alphabet $\Sigma$. There exists a bound $m > 0$ such that each $w \in L$ with $|w| \geqslant m$ can be written as $w = uvxyz$ with $u, v, x, y, z \in \Sigma^*$, $vy \neq \varepsilon$, $|vxy| \leqslant m$, and for all $k \geqslant 0$: $uv^k xy^k z \in L$.

*Proof.* Let $G = (V, \Sigma, R, S)$ be a context-free grammar that generates $L$. Let $\kappa = \max\{\, s \mid A \to X_1 \cdots X_s \in R \,\}$. Put $m = \kappa^{\#V+1}$.

Pick $w \in L$ with $|w| \geqslant m$. Let $PT$ be a parse tree for $w$, with root labeled $S$, and a minimal number of nodes.

If the height of $PT$ is $h$, then $|w| \leqslant \kappa^h$. Since $|w| \geqslant m = \kappa^{\#V+1}$ we have $h \geqslant \#V+1$. So $PT$ has a path from root to a leave of length $h$. This path has $h+1$ nodes of which $h$ nodes are labeled with a variable. Since $h \geqslant \#V+1$ there is a variable $A$ that occurs more than once on the path. See Figure 3.8.

Let $x$ be the yield of the parse tree $PT_{low}$ with the lowest occurrence of $A$ on the path as its root. Let $v$ and $y$ be such that $vxy$ is the yield of the parse tree $PT_{high}$ with the second to lowest occurrence of $A$ on the path as its root. Let $u$ and $z$ be such that $uvxyz$ is the yield of $PT$.

If we replace in $PT$ the subtree $PT_{high}$ by $PT_{low}$, we obtain a parse tree $PT'$ with root $S$ and yield $uxz$. Hence $uxz \in L$. If we replace in $PT$ the subtree $PT_{low}$ by $PT_{high}$, we obtain a parse tree with root $S$ and yield $uv^2xw^2z$. Hence $uv^2xw^2z \in L$. If we iterate this $k$ times we get a parse tree with root $S$ and yield $uv^kxw^kz$. Hence $uv^kxw^kz \in L$.

To prove that $vy \neq \varepsilon$ we argue that if $vy = \varepsilon$ then $v = \varepsilon$ and $y = \varepsilon$. Thus the parse tree $PT'$ above would be a parse tree with root $S$, yield $uxz = uvzyz = w$, but with a smaller number of nodes. This contradicts the minimality of $PT$. So $vy \neq \varepsilon$.

Finally, to establish that $|vxy| \leqslant m$ we argue that we can arrange that the height of parse tree $PT_{high}$ is at most $\#V + 1$ by taking the root node of $PT_{high}$ the lowest node for which there is a node labeled with the same variable down the path. It follows that $PT_{high}$ has at most $\kappa^{\#V+1} = m$ nodes. Since $vxy$ is the yield of $PT_{high}$, we have $|vxy| \leqslant m$. □

**Example 3.36.** The language $L = \{\, a^n b^n c^n \mid n \geqslant 0 \,\}$ is not context-free. We apply the Pumping Lemma. Choose $m > 0$ arbitrarily. Put $w = a^m b^m c^m \in L$. Then $|w| \geqslant m$. Suppose we $u, v, x, y, z$ are such that $w = uvxyz$, $vy \neq \varepsilon$. Since $|vy| \leqslant |vxy| \leqslant m$, the string $vy$ contains only $a$'s, only $a$'s and $b$'s, only $b$'s, only $b$'s and $c$'s, or only $c$'s. But then $w' = uv^2xy^2z$ has only more $a$'s, only more $a$'s and $b$'s, only more $b$'s, only more $b$'s and $c$'s, or only more $c$'s, respectively. So, the number of $a$'s, $b$'s and $c$'s in $w'$ is not in balance and $w' \notin L$. We conclude that there is no $m > 0$ that fulfills the requirements of the Pumping Lemma. Hence $L$ is not context-free.

**Example 3.37.** The language $L = \{\, a^n \mid n \text{ is prime} \,\}$ is not context-free. We apply the Pumping Lemma to show this: Choose $m > 0$ arbitrarily. Let $p$ be a prime number $p \geqslant m$. Put $w = a^p \in L$. Then $|w| \geqslant m$. Suppose we $u, v, x, y, z$ are such that $w = uvxyz$, $vy \neq \varepsilon$ and $uv^nxy^nz \in L$ for each $n \geqslant 0$. Put $q = |vy|$ and $r = |uxz|$. Note $q > 0$. We have that $nq + r$ is prime, for each $n \geqslant 0$. Consider $n = 2q + r + 2$.

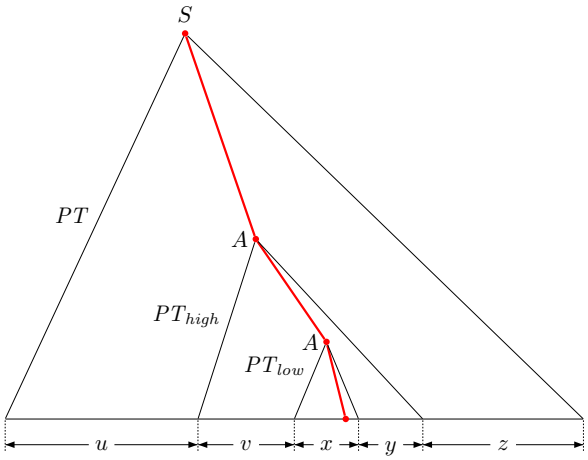



Figure 3.8: Repeating variable $A$ in minimal parse tree $PT$ with yield $uvxyz$.

It holds that $nq + r = (2q + r + 2)q + r = 2q^2 + rq + 2q + r = (q + 1)(2q + r)$. Since both $q + 1 > 1$ and $2q + r > 1$, it follows that $r + nq$ is not prime for this choice of $n$. Contradiction. So, no $m > 0$ meets the requirements of the Pumping Lemma. Hence $L$ is not a context-free language.

**Example 3.38.** Although $\{ ww^R \mid w \in \{a, b\}^* \}$ is a context-free language, the language $L = \{ ww \mid w \in \{a, b\}^* \}$ is not context-free. To see this, we apply the Pumping Lemma. Let $m > 0$ be arbitrary. Consider the string $a^m b^m a^m b^m \in L$. Pick $u, v, x, y, z$ such that $a^m b^m a^m b^m = uvxyz$, $vy \neq \varepsilon$, and $|vxy| \leqslant m$.

Since $|vxy| \leqslant m$ and $|vy| > 0$, leaving out $v$ and $y$ from $a^m b^m a^m b^m$ results in a string $uxz$ of the form

$$a^\ell b^m a^m b^m, \quad a^m b^\ell a^m b^m, \quad a^m b^m a^\ell b^m, \quad \text{or} \quad a^m b^m a^m b^\ell$$

where $0 \leqslant \ell < m$, or

$$a^\ell b^k a^m b^m, \quad a^m b^\ell a^k b^m, \quad \text{or} \quad a^m b^m a^\ell b^k$$

where $0 \leqslant k, \ell$, and $k < m$ or $\ell < m$. But, these possibilities cannot equal $ww$ for some string $w \in \{a, b\}^*$. This is because for $ww$ to match, $w$ must start with an $a$ and end in a $b$. Therefore, both the first group of $a$'s and $b$'s equals $w$ and the second group of $a$'s and $b$'s equals $w$. However, because of the restrictions on $\ell$ and $k$, in all cases this does not hold.

So no choice of $u, v, x, y, z$ can be right. Hence, no $m$ exists that meets the requirements of the Pumping Lemma. So, $L$ is not a context-free language.

**Example 3.39.** The language $L = \{ w \in \{a, b, c\}^* \mid \#_a(w) = \#_b(w) = \#_c(w) \}$ is not context-free. Suppose $L$ is context-free. Then, by Lemma 3.33, $L \cap a^* b^* c^*$ is context-free too. But $L \cap a^* b^* c^* = \{a^n b^n c^n \mid n \geqslant 0\}$ and is not context-free as shown in Example 3.36 above.

**Theorem 3.40.** The class of context-free languages is not closed under intersection and complement.

*Proof.* The languages $L_1 = \{ a^n b^n c^m \mid n, m \geqslant 0 \}$ and $L_2 = \{ a^n b^m c^m \mid n, m \geqslant 0 \}$ are both context-free. However, $L_1 \cap L_2 = \{ a^n b^n c^n \mid n \geqslant 0 \}$ which is not context-free according to Example 3.36.

The union of two context-free languages is context-free again, according to Theorem 3.32. If the complement $L^C$ of a context-free language $L$ would be context-free in general, then also

$$L_1 \cap L_2 = (L_1^C \cup L_2^C)^C$$

would be context-free. But it isn't, as was just shown. □

# Chapter 4

# Turing Machines and Computable Functions

We have seen that PDAs, containing a finite control and equipped with a stack-like memory, can accept a wide range of languages. The class of context-free languages can be characterized as the class of languages accepted by such a PDA. However, we have also seen some simple languages that are not context-free. The language $\{ a^n b^n c^n \mid n \geqslant 0 \}$ of Example 3.36 and the language $\{ ww \mid w \in \{a, b\}^* \}$ of Example 3.38 cannot be accepted by a PDA.

In this chapter, we consider reactive Turing machines like the machine depicted in Figure 4.1. Reactive Turing machines accept a class of languages called the recursively enumerable languages. This class is wider than the class of context-free languages accepted by PDAs. In particular, there are Turing machines that accept the languages of the examples mentioned above.

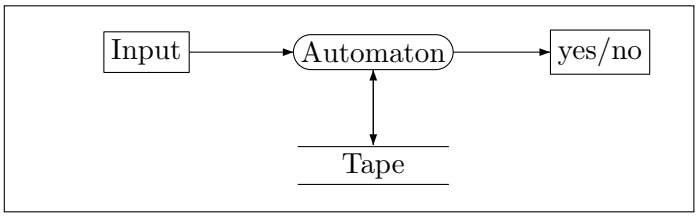



Figure 4.1: Reactive Turing machine.

The push-down automata of the previous chapter have a memory containing a stack of symbols which can only be accessed at the top. The Turing machine has a memory which again always has a string (or sequence of strings) as contents, but which now can be accessed at any given element of the string. This entails that the so-called tape head can move around the string which is the memory content. The Turing machine seems like a small advancement over the possibilities of a push-down automaton. Nevertheless, it can be argued that any computation that can be done by any computer can also be done by a Turing machine.

We will generalize the case of output of only "yes" or "no" to the case where the output is an arbitrary string at the end of a computation. In this case, we speak of a *computable* function.

## 4.1 The reactive Turing machine

We will first introduce some notation that is useful to describe Turing machines. The memory tape or just tape of a Turing machine consists of an unlimited number of sequentially ordered non-numbered cells, and each cell will contain a data element $d \in \Delta$ or is empty. We use the special symbol $\Box$, the *blank*, to denote an empty cell ($\Box \notin \Delta$). The extended tape alphabet $\Delta_\Box$, that we use besides the input alphabet $\Sigma$, consists of all symbols in $\Delta$ together with the blank, i.e. $\Delta_\Box = \Delta \cup \{\Box\}$.

At each point in an execution, the tape head of the Turing machine is positioned at exactly one particular cell, the cell in the eye of the tape head, of which it can read a data element or a blank and can write a data element or erase the contents (i.e. replace it with a blank). Then the tape head will move one cell to the right or one cell to the left.

We use the expression $x\langle e\rangle y$, with $x, y \in \Delta_\Box^*$, $e \in \Delta_\Box$, to denote a tape containing the string $xey$: the tape head is at a cell holding $e$, the cells left of the tape head together contain the string $x$, the cells right of the tape head together contain the string $y$. To define a unique expression $x\langle e\rangle y$ we require for $x \neq \varepsilon$ that the leftmost symbol of $x$ is not a blank. Thus, on the left, leading blanks are ignored. Likewise, if $y \neq \varepsilon$ then its rightmost symbol is assumed not to be a blank. Thus, on the right, trailing blanks are ignored. We write $\langle e\rangle$ as shorthand for $\varepsilon\langle e\rangle\varepsilon$, and $\langle e\rangle y$ and $x\langle e\rangle$ for $\varepsilon\langle e\rangle y$ and $x\langle e\rangle\varepsilon$, respectively. Note, $\langle\Box\rangle$ denotes the empty tape.

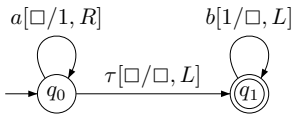



Figure 4.2: A simple reactive Turing machine.

**Example 4.1.** Consider the example in Figure 4.2, depicting a Turing machine with input alphabet $\Sigma = \{a, b\}$. The tape alphabet $\Delta = \{1\}$, hence $\Delta_\Box = \{1, \Box\}$. We start out in state $q_0$ with $\langle\Box\rangle$ in memory. Thus control is in the initial state and the tape head is reading a blank, also all other cells are empty. We can take the loop of $q_0$ if the symbol $a$ is on input; a 1 is written in the cell that is currently pointed at and the tape head moves to the right. Thus we obtain $1\langle\Box\rangle$ as tape content. Alternatively, we can take silently the edge labeled $\tau$ to the final state $q_1$, keeping $\langle\Box\rangle$ as it appears. (In fact, the tape head has moved to the left, but our notation doesn't show.) Taking the loop three times on input $aaa$ gives $111\langle\Box\rangle$ with control in state $q_0$, then taking the edge to $q_1$ gives $11\langle 1\rangle$, at which point, in $q_1$, we can either terminate, or execute the loop on

input $b$ to obtain $1\langle 1\rangle$. Maximally three $b$'s can be read at the expense of erasing a 1 for each $b$.

**Definition 4.2** (Reactive Turing machine)**.** A *reactive Turing machine*, or Turing machine for short, is a septuple $M = (\, Q,\, \Sigma,\, \Delta,\, \Box,\, \rightarrow,\, q_0,\, F\,)$ where

1. $Q$ is a finite set of states,
2. $\Sigma$ is a finite alphabet with $\tau \notin \Sigma$,
3. $\Delta$ is a finite data or tape alphabet, $\Box \notin \Delta$ a special symbol called *blank*,
4. $\rightarrow \subseteq Q \times \Sigma_\tau \times \Delta_\Box \times \Delta_\Box \times \{L, R\} \times Q$, where $\Sigma_\tau = \Sigma \cup \{\tau\}$ and $\Delta_\Box = \Delta \cup \{\Box\}$, is a finite set of *transitions* or *steps*,
5. $q_0 \in Q$ is the initial state,
6. $F \subseteq Q$ is the set of final states.

For $a \in \Sigma$, if $(q, a, e, e', \mu, q') \in \rightarrow$, we write $q \xrightarrow{a[e/e',\mu]}_M q'$, and this means that the Turing machine, when it is in state $q$, reading the symbol $a$ on input and reading the symbol $e$ on tape, it can consume the input, change the symbol on the tape to $e'$, move one cell left if $\mu = L$ and one cell right if $\mu = R$ and thereby change control to state $q'$. It is also possible that $e$ and/or $e'$ is $\Box$: if $e$ is $\Box$, we are looking at an empty cell on the tape; if $e'$ is $\Box$, then the symbol in the cell under the tape head (possibly a blank) will be erased.

Similarly, if $(q, \tau, e, e', \mu, q') \in \rightarrow$, we write $q \xrightarrow{\tau[e/e',\mu]}_M q'$. Now it means that the Turing machine, when it is in state $q$ and reading the symbol $e$ on tape, can (without a change on input) write the symbol $e'$ under the tape head, move left or right if $\mu = L$ or $\mu = R$, respectively, and change control to state $q'$. As a combined notion we may write $q \xrightarrow{\alpha[e/e',\mu]}_M q'$ with $\alpha$ ranging over $\Sigma_\tau$.

To process the steps of the Turing machine we need to have a means to update the tape. Suppose the tape contains $x\langle e\rangle y$ and we can execute the transition $q \xrightarrow{\alpha[e/e',\mu]}_M q'$. Note, the data symbol $e$ is both occurring in $x\langle e\rangle y$ as well is in the label $\alpha[e/e', \mu]$ of the transition. In this setting, we define the update $x\langle e\rangle y\,[e/e', \mu]$ of tape contents $x\langle e\rangle y$ with tape head action $[e/e', \mu]$ as follows.

- $x\langle e\rangle\varepsilon[e/e', R] = xe'\langle\Box\rangle\varepsilon$ for $x \in \Delta_\Box^*$,
- $x\langle e\rangle dy[e/e', R] = xe'\langle d\rangle y$ for $d \in \Delta_\Box$, $x, y \in \Delta_\Box^*$,
- $\varepsilon\langle e\rangle y[e/e', L] = \varepsilon\langle\Box\rangle e'y$ for $y \in \Delta_\Box^*$,

- $xd\langle e\rangle y[e/e', L] = x\langle d\rangle e'y$ for $d \in \Delta_\Box$, $x, y \in \Delta_\Box^*$.

Here in case the left string $x$ or right string $y$ are empty, we use the full notation $\varepsilon\langle e\rangle y$ and $x\langle e\rangle\varepsilon$, respectively. When using the shorthand, the empty string cases read $x\langle e\rangle\,[e/e', R] = xe'\langle\Box\rangle$ and $\langle e\rangle y\,[e/e', L] = \langle\Box\rangle e'y$.

We see that the contents of the currently scanned cell is filled with $e'$. If the tape head moves right the first symbol $d$ of $dy$ comes under the tape head or, in case of $\varepsilon$ a blank comes under the tape head. If the tape head moves left, two similar cases apply.

Expressions of the form $x\langle e\rangle y \in \Delta_\Box^* \times \Delta_\Box \times \Delta_\Box^*$ occur frequently in the following. We introduce the class of tape content $\mathcal{Z}$ by putting

$$\begin{aligned}\mathcal{Z} = \{\, x\langle e\rangle y \mid x \in \Delta_\Box^*\colon x = \varepsilon \vee \mathit{first}(x) \neq \Box, \\ e \in \Delta_\Box,\ y \in \Delta_\Box^*\colon y = \varepsilon \vee \mathit{last}(y) \neq \Box \,\}\end{aligned} \tag{4.1}$$

and have $z$ range over $\mathcal{Z}$. We want the first symbol of $x$ and the last symbol of $y$ to be a symbol in $\Sigma$. This way, given the position of the tape head, the tape content is uniquely defined.

With the notion of an update available we can define a configuration or instantaneous description of a Turing machine $M = (\,Q,\, \Sigma,\, \Delta,\, \Box,\, \to,\, q_0,\, F\,)$, ID for short. A configuration of the Turing machine $M$ is a triple $(q, w, z)$ of a state $q$, an input string $w$ and a tape content $z$. Thus $(q, w, z) \in Q \times \Sigma^* \times \mathcal{Z}$. The $q$ and $z$, respectively, are the current state and current content of the tape, $w$ represents the input that is not read so far.

We write $(q, w, z) \vdash_M (q', w', z')$ iff either (i) for some $a \in \Sigma_\tau$, $e, e' \in \Delta_\Box$, and $\mu \in \{L, R\}$ we have

$$q \xrightarrow{a[e/e',\mu]}_M q' \ \wedge\ w = aw' \ \wedge\ z[e/e', \mu] = z'$$

or, (ii) for some $e, e' \in \Delta_\Box$, and $\mu \in \{L, R\}$ we have

$$q \xrightarrow{\tau[e/e',\mu]}_M q' \ \wedge\ w = w' \ \wedge\ z[e/e', \mu] = z'$$

Thus, we have $(q, w, z) \vdash_M (q', w', z')$ if Turing machine $M$ can move from configuration $(q, w, z)$ in one step to configuration $(q', w', z')$. Note, this does not exclude that $(q, w, z) \vdash_M (q'', w'', z'')$ for a configuration $(q'', w'', z'')$ different from $(q', w', z')$. We write $(q, w, z) \nvdash_M$ if for no $q'$, $w'$ and $z'$ we have $(q, w, z) \vdash_M (q', w', z')$. In such a situation we say that the Turing machine $M$ blocks or halts.

At the start of an execution of a reactive Turing machine, we will assume the Turing machine is in the initial state, and that the memory tape is empty. Thus, the initial configuration of a reactive Turing machine with initial state $q_0$ and the string $w$ on input is $(q_0, w, \langle\Box\rangle)$.

**Example 4.3.** Consider the Turing machine depicted in Figure 4.3 with input alphabet $\Sigma = \{a, b\}$, tape alphabet $\Delta = \{1\}$ and blank $\Box$. When started in $q_0$, any number of $a$'s can be input, each time writing the symbol 1 on the tape and moving the tape head to

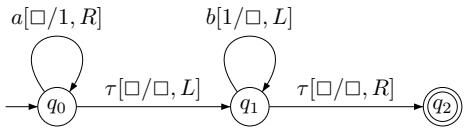



Figure 4.3: Another simple reactive Turing machine.

the right. At any time, a move to the state $q_1$ can occur, reversing the direction of the tape head. Then, the same number of $b$'s can be input, each time erasing a 1. Coming to the beginning of the string of 1's while having erased these in transit, the tape is empty again. The tape head is scanning a blank and termination can take place after a silent transition from state $q_1$ to $q_2$.

A computation showing that $M$ accepts the string $aaabbb$ is the following:

$$(q_0, aaabbb, \langle\Box\rangle) \vdash_M (q_0, aabbb, 1\langle\Box\rangle) \vdash_M (q_0, abbb, 11\langle\Box\rangle) \vdash_M (q_0, bbb, 111\langle\Box\rangle) \vdash_M$$
$$(q_1, bbb, 11\langle 1\rangle) \vdash_M (q_1, bb, 1\langle 1\rangle) \vdash_M (q_1, b, \langle 1\rangle) \vdash_M (q_1, \varepsilon, \langle\Box\rangle) \vdash_M (q_2, \varepsilon, \langle\Box\rangle)$$

The language of the Turing machine of Figure 4.3, a notion to be formally defined below, is the set of strings $\{\, a^n b^n \mid n \geqslant 0\}$.

**Definition 4.4.** Let $M = (\, Q, \Sigma, \Delta, \Box, \rightarrow, q_0, F\,)$ be a Turing machine. Then the language $\mathcal{L}(M) \subseteq \Sigma^*$ accepted by $M$ is defined by

$$\mathcal{L}(M) = \{\, w \in \Sigma^* \mid \exists q \in F \exists z \in \mathcal{Z} \colon (q_0, w, \langle\Box\rangle) \vdash_M^* (q, \varepsilon, z)\,\}$$

In the definition above, $\vdash_M^*$ denotes the reflexive and transitive closure of the relation $\vdash_M$. Thus

$$\begin{aligned}
&(q, w, z) \vdash_M^* (q', w', z') \quad \text{iff} \\
&\quad \exists n \geqslant 0\, \exists q_0, \ldots, q_n\, \exists w_0, \ldots, w_n\, \exists z_0, \ldots, z_n : \\
&\qquad (q_0, w_0, z_0) = (q, w, z) \wedge \\
&\qquad\quad \forall i,\, 1 \leqslant i \leqslant n \colon (q_{i-1}, w_{i-1}, z_{i-1}) \vdash_M (q_i, w_i, z_i) \wedge \\
&\qquad\qquad (q_n, w_n, z_n) = (q', w', z')
\end{aligned}$$

For a number of languages, Turing machines can be constructed. We start out easy.

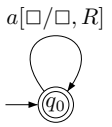



Figure 4.4: A reactive Turing machine for $\{a\}^*$.

**Example 4.5.** A Turing machine that accepts the language $\{a\}^* = \{\, a^n \mid n \geqslant 0 \,\}$ is shown in Figure 4.4. As long as the input consists of $a$'s, we move to the right. At any time, termination can occur. Although the tape head may move, not data symbol is written on tape.

**Example 4.6.** Every finite language is accepted by a Turing machine. For example the finite language $\{a, ba, cba\}$ over $\{a, b, c\}$ is accepted by the Turing machine given in Figure 4.5.

In fact, a stronger result than Example 4.6 holds: for every regular language $L$, there exists a Turing machine that accepts $L$. Thus the class of regular languages is contained in the class of recursive enumerable languages.

**Theorem 4.7.** Let $L \subseteq \Sigma^*$ be a language such that $L = \mathcal{L}(D)$ for a deterministic finite automaton $D$, then there exists a Turing machine $M$ such that $L = \mathcal{L}(M)$.

*Proof.* Suppose $D = (\, Q,\, \Sigma,\, \delta,\, q_0,\, F,\, )$ is a DFA that accepts $L$. Define the Turing machine $M = (\, Q,\, \Sigma,\, \emptyset,\, \square,\, \rightarrow,\, q_0,\, F)$ with the same set of states, the same input alphabet, the empty tape alphabet, the standard blank, the same initial state and the same set of final states and a transition relation $\rightarrow$ given by

$$\rightarrow \;=\; \{\, (q, a, \square, \square, R, q') \mid \delta(q, a) = q' \,\}$$

Thus, the transitions $q \xrightarrow{a[\square/\square,R]} \delta(q, a)$, for $q \in Q$, $a \in \Sigma$, are the only transitions of $M$. Note, since the tape alphabet of $M$ is the empty set no other symbol than a blank can be written on tape. So, the tape will remain empty, i.e. all tape cells will contain a blank, for all computations of $M$.

By definition of $\rightarrow$ it holds that

$$\begin{aligned}
&(q, w) \vdash_D (q', w') \\
&\quad\Leftrightarrow\quad \exists a : w = aw' \wedge \delta(q, a) = q' \\
&\quad\Leftrightarrow\quad \exists a : w = aw' \wedge q \xrightarrow{a[\square/\square,R]} q' \\
&\quad\Leftrightarrow\quad (q, w, \langle\square\rangle) \vdash_M (q', w', \langle\square\rangle)
\end{aligned}$$

Thus it also holds, as can be shown by induction, that

$$(q_0, w) \vdash_D^* (q, \varepsilon) \iff (q_0, w, \langle\square\rangle) \vdash_M^* (q, \varepsilon, \langle\square\rangle)$$

From this it follows that $\mathcal{L}(M) = \mathcal{L}(D)$, which proves the theorem. □

Note, since the language $\{\, a^n b^n \mid n \geqslant 0 \,\}$ is a recursively enumerable language as we have seen in Example 4.3, it follows from the theorem that the class of recursively enumerable languages is strictly larger than the class of regular languages.

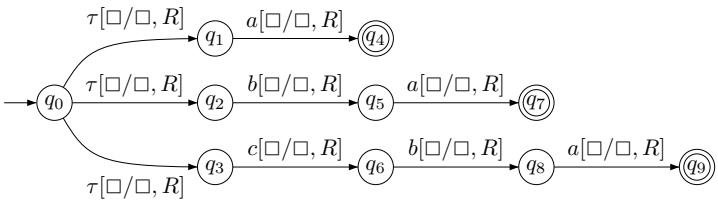



Figure 4.5: A reactive Turing machine for $\{a, ba, cba\}$.

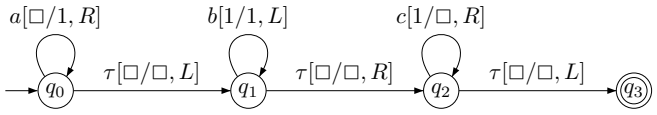



Figure 4.6: A reactive Turing machine for $\{\, a^n b^n c^n \mid n \geqslant 0 \,\}$.

**Example 4.8.** We have seen that the language $\{\, a^n b^n c^n \mid n \geqslant 0 \,\}$ is not a context-free language. However, an extension of the simple Turing machine of Figure 4.3 accepts this language. See Figure 4.6. Together with Theorem 4.10 below, this implies that the class of languages accepted by a Turing machine, i.e. the recursively enumerable languages, is strictly larger than the class of context-free languages.

**Example 4.9.** With a variation on the Turing machine in Figure 4.6, also a Turing machine for the language $\{\, ww \mid w \in \{a, b\}^* \,\}$ can be constructed. In the initial state $q_0$ for each $a$ and $b$ on input an $A$ or $B$ is written on tape. The Turing machine can non-deterministically go from state $q_0$ to state $q_1$. There the tape head moves left to the first blank left of the sequence of $A$'s and $B$'s. The Turing machine goes to state $q_2$. Then a sequence of $a$'s and $b$'s is input and it is checked that it matches the sequence of $A$'s and $B$'s on the tape. If the match is exact and all input is read, the input string is accepted. If the match is not exact the Turing machine blocks.

We have $(q_0, abaaba, \langle\Box\rangle) \vdash^* (q_3, \varepsilon, \langle\Box\rangle)$ since

$$
\begin{array}{lll}
(q_0, abaaba, \langle\Box\rangle) & & \\
\quad \vdash & (q_0, baaba, A\langle\Box\rangle) & \vdash \quad (q_0, aaba, AB\langle\Box\rangle) \\
\quad \vdash & (q_0, aba, ABA\langle\Box\rangle) & \vdash \quad (q_1, aba, AB\langle A\rangle) \\
\quad \vdash & (q_1, aba, A\langle B\rangle A) & \vdash \quad (q_1, aba, \langle A\rangle BA) \\
\quad \vdash & (q_1, aba, \langle\Box\rangle ABA) & \vdash \quad (q_2, aba, \langle A\rangle BA) \\
\quad \vdash & (q_2, ba, \langle B\rangle A) & \vdash \quad (q_2, a, \langle A\rangle) \\
\quad \vdash & (q_2, \varepsilon, \langle\Box\rangle) & \vdash \quad (q_3, \varepsilon, \langle\Box\rangle)
\end{array}
$$

Thus, indeed, the string $abaaba$ is accepted by the Turing machine. There are many

more computations for the configuration $(q_0, abaaba, \langle\Box\rangle)$, e.g.

$$\begin{aligned}
(q_0, abaaba, \langle\Box\rangle) &\vdash^* (q_0, ba, ABAA\langle\Box\rangle),\\
(q_0, abaaba, \langle\Box\rangle) &\vdash^* (q_0, a, ABAAB\langle\Box\rangle),\\
(q_0, abaaba, \langle\Box\rangle) &\vdash^* (q_1, a, ABAA\langle B\rangle),\\
(q_0, abaaba, \langle\Box\rangle) &\vdash^* (q_2, a, \langle A\rangle BAAB),\\
(q_0, abaaba, \langle\Box\rangle) &\vdash^* (q_2, \varepsilon, \langle B\rangle AAB),
\end{aligned}$$

but these do not lead to the accepting state $q_3$.

The configuration $(q_0, abaabab, \langle\Box\rangle)$ does have a derivation sequence to $q_3$. It holds that $(q_0, abaabab, \langle\Box\rangle) \vdash^* (q_3, b, \langle\Box\rangle)$. However, not all input has been read; the rightmost symbol $b$ is not processed. Therefore, the string $abaabab$ is *not* accepted by the Turing machine.

The crucial point is the non-determinism in state $q_0$. With a blank at the tape head and a symbol $a$ or $b$ on input, either the input symbol can be processed by the transition looping on $q_0$, or a silent step can be taken leading to the state $q_1$.

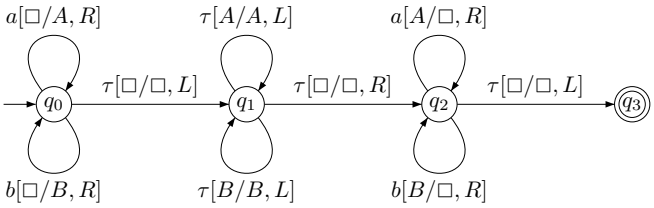



Figure 4.7: A reactive Turing machine for $\{\, ww \mid w \in \{a, b\}^* \,\}$.

Next we prove a result for context-free languages similar to Theorem 4.7 for regular languages.

**Theorem 4.10.** Let $L \subseteq \Sigma^*$ be a language such that $L = \mathcal{L}(P)$ for a push-down automaton $P$, then there exists a Turing machine $M$ such that $L = \mathcal{L}(M)$.

*Proof.* Let $P = (\, Q,\, \Sigma,\, \Delta,\, \emptyset,\, \rightarrow_P,\, q_0,\, F\,)$ be a push-down automaton accepting $L$. Define the Turing machine $M = (\, Q',\, \Sigma,\, \Delta_\emptyset,\, \rightarrow_M,\, q_0',\, F\,)$ with an extended set of states $Q' \supseteq Q$, the same input alphabet, the same tape alphabet to which the empty stack symbol $\emptyset$ is added, a new initial state, the same set of final states,

$$\begin{aligned}
Q' = Q \cup \{q_0', q_0''\} \cup \{\, [alloc, x, q'] \mid \exists q \in Q \exists \alpha \in \Sigma_\tau \exists d \in \Delta_\emptyset : q \xrightarrow{\alpha[d/x]}_P q' \,\} \cup\\
\{\, [alloc, x', x, q'] \mid \exists q \in Q \exists \alpha \in \Sigma_\tau \exists d \in \Delta : q \xrightarrow{\alpha[d/x]}_P q' \wedge x' \text{ is suffix of } x \,\} \cup\\
\{\, [write, x', q'] \mid \exists q \in Q \exists \alpha \in \Sigma_\tau \exists d \in \Delta_\emptyset \exists x \in \Delta^* : q \xrightarrow{\alpha[d/x]}_P q' \wedge x' \text{ is suffix of } x \,\} \cup\\
\{\, [seek, q'] \mid \exists q \in Q \exists \alpha \in \Sigma_\tau \exists d \in \Delta_\emptyset \exists x \in \Delta^* : q \xrightarrow{\alpha[d/x]}_P q' \,\}
\end{aligned}$$

for two fresh states $q_0', q_0'' \notin Q$, for each transition $q \xrightarrow{\alpha[d/x]}_P q'$, with $q, q' \in Q$, $\alpha \in \Sigma_\tau$, $d \in \Delta$ and $x \in \Delta^*$, and we have

$$\begin{array}{rl} q & \xrightarrow{\alpha[d/\Box,L]}_M \quad [alloc, x, q'] \\ [alloc, x, q'] & \xrightarrow{\tau[\Box/\Box,R]}_M \quad [alloc, x, x, q'] \end{array}$$

for each transition $q \xrightarrow{\alpha[\emptyset/x]}_P q'$, with $q, q' \in Q$, $\alpha \in \Sigma_\tau$, and $x \in \Delta^*$, we have

$$q \xrightarrow{\alpha[\emptyset/\emptyset,L]}_M [alloc, x, x, q']$$

Moreover, for each transition $q \xrightarrow{\alpha[d/x]}_P q'$, with $q, q' \in Q$, $\alpha \in \Sigma_\tau$, $d \in \Delta_\emptyset$, $x \in \Delta^*$, and $d'x'$ a suffix of $x$ for $d' \in \Delta$, $x' \in \Delta^*$, $e \in \Delta_\emptyset$, $e' \in \Delta$, we have

$$\begin{array}{rl} [alloc, \varepsilon, x, q'] & \xrightarrow{\tau[\Box/\Box,R]}_M \quad [write, x, q'] \\ [alloc, d'x', x, q'] & \xrightarrow{\tau[\Box/\Box,L]}_M \quad [alloc, x', x, q'] \\ [write, \varepsilon, q'] & \xrightarrow{\tau[e/e,L]}_M \quad [seek, q'] \\ [write, d'x', q'] & \xrightarrow{\tau[\Box/d',R]}_M \quad [write, x', q'] \\ [seek, q'] & \xrightarrow{\tau[e'/e',L]}_M \quad [seek, q'] \\ [seek, q'] & \xrightarrow{\tau[\Box/\Box,R]}_M \quad q' \end{array}$$

and finally,

$$\begin{array}{rl} q_0' & \xrightarrow{\tau[\Box/\emptyset,L]}_M \quad q_0'' \\ q_0'' & \xrightarrow{\tau[\Box/\Box,R]}_M \quad q_0 \end{array}$$

Now, define the tape content function $tc : \Delta^* \to \mathcal{Z}$ by

$$tc(\varepsilon) = \langle\emptyset\rangle \quad \text{and} \quad tc(dx) = \langle d\rangle x\emptyset$$

Then it can be shown by induction that

$$(q, w, x) \vdash_P (q', w', x') \iff (q, w, tc(x)) \vdash_M^* (q', w', tc(x'))$$

for all $q, q' \in Q$, $w, w' \in \Sigma^*$ and $x, x' \in \Delta^*$. From this it follows that

$$(q, w, x) \vdash_P^* (q', w', x') \iff (q, w, tc(x)) \vdash_M^* (q', w', tc(x'))$$

In particular, for $q' \in F$,

$$(q_0, w, \varepsilon) \vdash_P^* (q', \varepsilon, x') \iff (q_0, w, \langle\emptyset\rangle) \vdash_M^* (q', \varepsilon, tc(x'))$$

Thus, since $(q_0', \varepsilon, \langle\Box\rangle) \vdash_M^* (q_0, \varepsilon, \langle\emptyset\rangle)$, we conclude $\mathcal{L}(P) = \mathcal{L}(M)$. □

From the theorem it follows that the class of context-free languages is a subset of the class of recursively enumerable languages. In view of Examples 4.8 and 4.9 this inclusion is strict.

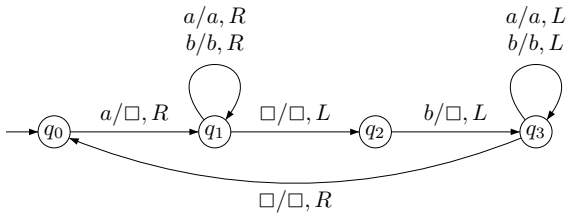



Figure 4.8: A classical Turing machine.

## 4.2 The classical Turing machine

The reactive Turing machine introduced in the previous section is used as language acceptor. However, with some conventions and simplifications a Turing machine can be used to compute a function too. The latter variant of a Turing machine is referred to as the classical Turing machine.

**Definition 4.11** (Classical Turing machine)**.** A classical Turing machine is a quintuple $M = (\, Q,\, \Delta,\, \square,\, \rightarrow,\, q_0\,)$ where $Q$, $\Delta$, $\square$, and $q_0$ are as before (see Definition 4.2), and

$$\rightarrow \subseteq Q \times \Delta_\square \times \Delta_\square \times \{L, R\} \times Q$$

with $\Delta_\square = \Delta \cup \{\square\}$.

If, for a Turing machine $M$ as given above, $(q, e, e', \mu, q') \in \rightarrow$ we write $q \xrightarrow{e/e',\mu}_M q'$. This means that the Turing machine, when it is in state $q$ and reads symbol $e$ on the tape, can replace $e$ by $e'$, move one cell left if $\mu = L$ and one cell right if $\mu = R$, and thereby change control to state $q'$. There is no input alphabet $\Sigma$ as for the reactive Turing machine; there are no $\tau$-transitions either. Also, there are no final states. For a classical Turing machine there is no notion of acceptance, but there is a notion of termination as we will see below.

**Example 4.12.** An example of a classical Turing $M$ is given in Figure 4.8. We have $\Delta = \{a, b\}$. Note that the Turing machine has no transition for the symbol $b$ in state $q_0$ and no transition for the symbols $a$ and $\square$ in state $q_2$. Thus, if in state $q_0$ the symbol $b$ is read on tape, the Turing machine $M$ blocks and the computation halts. Similarly, if in state $q_2$ either the symbol $a$ or a blank is read, the Turing machine $M$ blocks.

Suppose $M$ starts in state $q_0$, with the string *aaabbb* written on tape and with the tape head on the leftmost $a$. Then $M$ erases this $a$, the tape head moves to the right, control to state $q_1$. Then the string *aabbb* is skipped while the tape head moves to the right, and $M$ reads a blank right after the string of $a$'s and $b$'s. $M$ moves to the left and arrives in state $q_2$. Then the rightmost $b$ is read, erased, the tape head moves left, control to state $q_3$. Then the Turing machine skips over the remaining string *aabb*, till it reads the first blank on the left (where the cell where the tape head started from initially). Then the tape head moves right reading the symbol $a$, control is now in state $q_0$.

To describe the behaviour concisely, we adapt the notion of a configuration to the situation of a classical Turing machine. Let $M = (\, Q,\, \Delta,\, \Box,\, \rightarrow,\, q_0\,)$ be a classical Turing machine according to Definition 4.11. A configuration or an ID of $M$ is a pair $(q, z) \in Q \times \mathcal{Z}$ with $\mathcal{Z}$ as defined in the previous section, see Equation (4.1). We write $(q, z) \vdash_M (q', z')$ iff for some $e, e' \in \Delta_\Box$ and $\mu \in \{L, R\}$ we have

$$q \xrightarrow{e/e',\mu}_M q' \quad \text{and} \quad z' = z[e/e', \mu]$$

The operation $[e/e', \mu]$, for $e, e' \in \Delta_\Box$ and $\mu \in \{L, R\}$, is as before. Here, for $z' = z[e/e', \mu]$, the tape content $z'$ is obtained from $z$ by replacing the eye $e$ of $z$ by $e'$ and moving the tape head in the direction indicated by $\mu$. Note that this replacement requires that $z$ is of the form $x\langle e\rangle y$; otherwise the operation is not defined.

We write $(q, z) \nvdash_M$ if for no $q'$ and $z'$ we have $(q, z) \vdash_M (q', z')$. If $z = x\langle e\rangle y$ and there is no transition $q \xrightarrow{e/e',\mu}_M q'$ with $q' \in Q$, $e' \in \Delta_\Box$ and $\mu \in \{L, R\}$ , we say that $M$ halts or blocks in $(q, z)$. As there is no further transition, any computation leading to the configuration $(q, z)$ terminates there.

For the classical Turing machine $M$ of Example 4.12 we have the following computation:

$$\begin{aligned}
&(q_0, \langle a\rangle aabbb) \vdash_M (q_1, \langle a\rangle abbb) \vdash_M (q_1, a\langle a\rangle bbb) \vdash_M (q_1, aa\langle b\rangle bb) \vdash_M \\
&\quad (q_1, aab\langle b\rangle b) \vdash_M (q_1, aabb\langle b\rangle) \vdash_M (q_1, aabbb\langle\Box\rangle) \vdash_M (q_2, aabb\langle b\rangle) \vdash_M \\
&\quad\quad (q_3, aab\langle b\rangle) \vdash_M (q_3, aa\langle b\rangle b) \vdash_M (q_3, a\langle a\rangle bb) \vdash_M (q_3, \langle a\rangle abb) \vdash_M \\
&\quad\quad\quad (q_3, \langle\Box\rangle aabb) \vdash_M (q_0, \langle a\rangle abb)
\end{aligned}$$

Thus $(q_0, \langle a\rangle aabbb) \vdash_M^* (q_0, \langle a\rangle abb)$. However, $M$ does not halt in $(q_0, \langle a\rangle abb)$, since $(q_0, \langle a\rangle abb)$ admits a further transition, viz. to $(q_1, \langle a\rangle bb)$. The computation is not complete.

It also holds that $(q_0, \langle a\rangle aabbb) \vdash_M^* (q_0, \langle\Box\rangle)$. Since $(q_0, \langle\Box\rangle) \nvdash_M$ the computation started in the configuration $(q_0, \langle a\rangle aabbb)$ is complete. It terminates in the configuration $(q_0, \langle\Box\rangle)$.

We are about to introduce the notion of a function *computed* by a classical Turing machine. However, we first need a means to relate tape content and strings.

We will use the notation $\langle w\rangle$, for a string $w \in \Delta^*$, to denote a tape containing $w$ and with the tape head at the leftmost symbol of $w$, if available. Formally, $\langle\varepsilon\rangle = \langle\Box\rangle$ and $\langle dw\rangle = \langle d\rangle w$. Thus, we have $\langle\cdot\rangle : \Delta^* \to \mathcal{Z}$. Note, for a tape content $z$ and non-empty string $w$, if $z = \langle w\rangle$ then $z$ contains a consecutive block of symbols from $\Delta$ comprising $w$, the tape is empty everywhere else. If $w = \varepsilon$ is the empty string, $\langle\varepsilon\rangle = \langle\Box\rangle$ represents the empty tape.

**Definition 4.13.** A classical Turing machine $M = (\, Q,\, \Delta,\, \Box,\, \rightarrow,\, q_0\,)$ computes a function $f : \Sigma^* \to \Theta^*$ for two alphabets $\Sigma$ and $\Theta$ $(\Sigma \cup \Theta \subseteq \Delta)$ iff, for all strings $w \in \Sigma^*$, we have

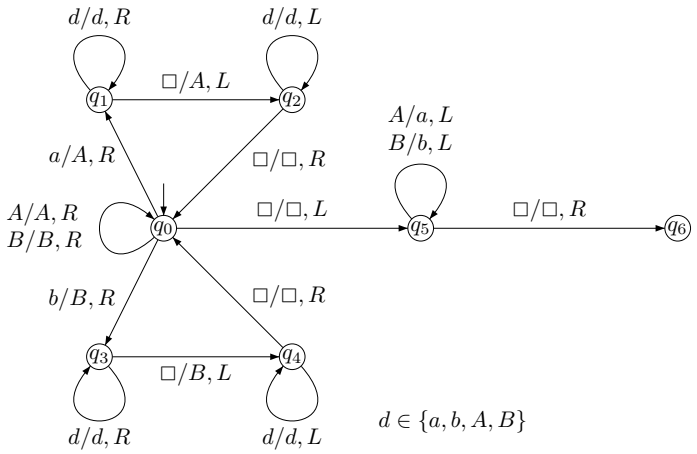



Figure 4.9: Classical Turing machine for the copying function $f(w) = ww$.

- $(q_0, \langle w \rangle) \vdash_M (q, z) \nvdash_M$ for some $q \in Q$, $z \in \mathcal{Z}$ with $z = \langle f(w) \rangle$;
- if $(q_0, \langle w \rangle) \vdash_M (q_1, z_1) \nvdash_M$ and $(q_0, \langle w \rangle) \vdash_M (q_2, z_2) \nvdash_M$, for $q_1, q_2 \in Q$, $z_1, z_2 \in \mathcal{Z}$, then $q_1 = q_2$ and $z_1 = z_2$.

For the classical Turing machine $M$ to compute for a string $w \in \Sigma^*$ the function value $f(w) \in \Theta^*$ for the string $w \in \Sigma^*$, the scheme is to first write the string $w$ on tape and start the Turing machine in its initial state $q_0$ with the tape head on the leftmost symbol of $w$, i.e. in the configuration $(q_0, \langle w \rangle)$. When the Turing machine terminates, say in the configuration $(q, z)$, the tape should contain a unique string from $\Theta^*$, called $f(w)$. Moreover the tape head is supposed to be at the leftmost symbol of $f(w)$ is it is non-empty.

For this to work, there must be at least one terminating computation for $M$ starting from $(q_0, \langle w \rangle)$ which yields a result in $\Theta^*$. Moreover, every terminating computation should yield the same result. The former is captured by the first condition of the definition. Regarding the latter, this condition is guaranteed if the transition relation $\rightarrow_M$ represents a partial function $Q \times \Delta_\square \rightarrow \Delta_\square \times \{L, R\} \times Q$, i.e. for every $q \in Q$, $e \in \Delta_\square$ there is at most one triple $e' \in \Delta_\square$, $\mu \in \{L, R\}$, $q' \in Q$ such that $q \xrightarrow{e/e', \mu}_M q'$.

**Example 4.14.** Figure 4.9 describes a classical Turing machine, say $M$, that computes the copying function $copy : \{a, b\}^* \rightarrow \{a, b\}^*$ with $copy(w) = ww$. The trick is to use the auxiliary symbols $A$ and $B$ to describe $a$'s and $b$'s that are already processed. Thus we have $\Delta = \{a, b, A, B\}$. However $\Sigma = \Theta = \{a, b\}$. The computation in $q_0$ starts on the leftmost non-blank symbol on the tape. Then $M$ scans to the right for the first $a$ or $b$. If there is none, the copying is done and $M$ changes control to state $q_5$ where is replaces all $A$'s and $B$'s by $a$'s and $b$'s, respectively. If a blank is in the eye of the tape head, it moves right and then the Turing machine blocks.

If in $q_0$ an $a$ is read, this symbol needs to be copied at the end of the tape content. The current $a$ is marked as processed, i.e. replaced by $A$. In state $q_1$ all non-blanks are

skipped, $d/d, R$ abbreviates $a/a, R$, $b/b, R$, $A/A, R$ and $B/B, R$. When the blank at the end of the tape content is reached, an $A$ is written and control is in state $q_2$. There the tape head is moved to the left of the tape content. When reached, $M$ is in state $q_0$ with the tape head on the leftmost non-blank symbol (if any).

If in $q_0$ a $b$ is read, a similar sequence of transitions follows. Now the symbol $b$ is overwritten with $B$, control changes to state $q_3$ and a $B$ is written at the end of the tape content, after which the tape head moves the leftmost symbol on tape and control moves to state $q_0$.

Dependent on the input string $w$, for suitable $U \in \{A, B\}^*$, we have

$$(q_0, U\langle a\rangle vU) \vdash_M^* (q_1, UAvU\langle\Box\rangle) \quad \text{and} \quad (q_0, U\langle b\rangle vU) \vdash_M^* (q_3, UBvU\langle\Box\rangle)$$

Functions with more than one argument can be computed too with a classical Turing machine. For example, for a two-ary function $f(w_1, w_2)$, that takes two strings $w_1$ and $w_2$ to produce a result, we put both strings on tape, first $w_1$ then $w_2$, have them separated by a blank, or an other symbol, and have the tape head positioned at the first, i.e., leftmost symbol of $w_1$ (or at the separating symbol if $w_1 = \varepsilon$).

**Example 4.15.** As a first example of a function computing with numbers we consider subtraction for non-negative integer numbers in unary notation. In unary notation a number $n$ is represented by the string $1^n$, i.e. a string of $n$ times the symbol 1. For simplicity, we assume that the two arguments are given as positive integers. As discussed above, we choose the numbers to be separated by a minus sign '$-$'. Thus the input to compute $n - m$ is the string $1^n{-}1^m \in \{1, \Box\}^*$ with $n, m \geqslant 1$. As we are dealing with non-negative numbers the convention is that the result is 0 if $n \leqslant m$.

The classical Turing machine of Figure 4.10 can be dissected in four parts: The path from the initial state $q_0$ up to state $q_2$, the cycle of state $q_2$ via state $q_5$, and the two paths to a state without outgoing transitions, viz. from state $q_5$ to state $q_9$ and from state $q_2$ to state $q_{11}$.

Starting in state $q_0$ the first concern is to find the right end of the arguments. So, we first skip to the right over the first number in state $q_0$, skip over the minus sign separating the two numbers, and skip over the second number in state $q_1$.

The loop of state $q_2$ visiting $q_3$, $q_4$, $q_5$, $q_6$ and $q_7$, respectively, is erasing a 1-symbol from the second number against a 1-symbol of the first number. This subcomputation starts at the rightmost symbol of the second number. In $q_2$ if a 1-symbol is read it is erased, the tape head skips to the left, skipping the remaining 1's of the second number in $q_3$, skipping the separator $-$, skipping the remaining 1's of the first number in $q_4$ till the blank at the left is found. The tape head moves one position back, control is state $q_5$. If a matching 1-symbol of the first number is read, this symbol is erased, and the tape head skips to the right, reaching via $q_6$, where the remaining 1's of the first number are skipped, and via $q_7$ where the remaining 1's of the second number are skipped the start of the cycle, state $q_2$ again with the tape head on the rightmost symbol, if any.

However, if in state $q_5$ no matching 1 is found, i.e. the tape head reads the separator rather than a 1, the result of the computation should be zero, since the second

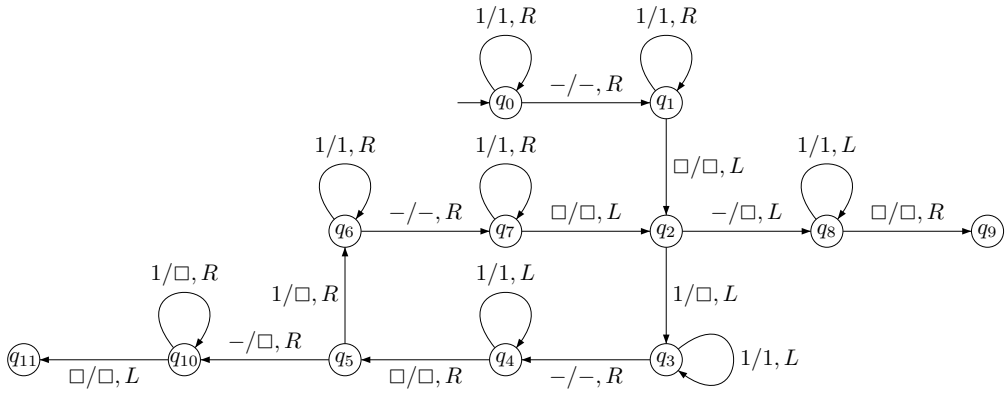



Figure 4.10: Classical Turing machine for unary subtraction.

number proves larger than the first number. The Turing machine is in configuration $(q_5, \langle - \rangle 1^{m-n-1})$. Note, in this case $m > n$. So next, all 1 are swiped from the tape in state $q_{10}$. This stops when a blank is found. The computation terminates and the tape is empty, representing $1^0 = \varepsilon$.

Alternatively, in state $q_2$ if no 1 is read any more, thus we are reading the separator $-$ rather than a 1-symbol of the second number, we are basically done. The configuration of the Turing machine is $(q_2, 1^{n-m}\langle - \rangle)$. We first need to clean up the separator, as this is no part of the output, and then move the tape head to the leftmost 1 (if any). The computation terminates leaving the result $1^{n-m}$ on tape.

**Example 4.16.** A Turing machine computing addition of binary numbers is depicted in Figure 4.11. For simplicity, we assume the two numbers that are to be added to have an equal number of digits, padding leading 0's to the shortest number if necessary. The numbers are separated by a plus sign '+'. The tape alphabet is the set $\Delta = \{\, 0,\, 1,\, +,\, Z,\, W,\, T \,\}$. The digits 0 and 1 as well as the $+$-sign speak for themselves. The capitals $Z$, $W$ and $T$ represent processed digits: $Z$ for 0 (zero), $W$ for 1 (one) (to avoid confusion between the digit 0 and the capital $O$ we use $W$ instead), $T$ for 2 (two) being the sum of two digits 1.

The Turing machine computes a function $+ : \Delta^* \to \Delta^*$. Only in case $w$ is a string of the form $w_1 + w_2$ with $w_1, w_2 \in \{0,1\}^n$ for some $n \geqslant 1$, we assure that the binary representation of the number $w_1 + w_2$ is computed. For conciseness, in Figure 4.11 we use the symbol $d$ to range over $0, 1$, the symbol $D$ to range over $Z, W, T$, and the symbol $e$ to range over all non-blanks.

- Starting in $q_0$ with the tape head at the leftmost digit of the first number, the tape head is moved to the blank right of the second number. In state $q_1$ the tape head is positioned at the rightmost digit of the second number.

- If in $q_1$ the digit 0 of the second number is read, the idea is to change the corresponding digit of the first number into $Z$ or $W$, dependent of the value of the latter

digit. To this end the tape head wobbles to the left. Skipping over 0's and 1's first in state $q_2$, then encountering a $+$, then skipping marked digits, i.e. capitals, in state $q_3$ and encountering the first digit to the left of the $+$. The digit 0 is marked as $Z$; we have read a 0 of the second number and $0+0=Z$. The digit 1 is marked as $W$; we have a read a 0 of the second number and $1+0=W$. In state $q_4$ the tape head moves to the right in search of the rightmost blank. Once found, the tape head is positioned on the rightmost digit of the second number. Control is in state $q_1$.

- If in $q_1$ the digit 1 of the second number is read, the Turing machine will change the corresponding digit of the first number into $W$ or $T$, again dependent on the value of the latter digit. In state $q_5$ the tape head moves left over the remainder of the second number, changing control to state $q_6$ if the $+$-sign is scanned. In state $q_6$ capitals are skipped until the first digit on the left of the $+$-sign is found. If it is the digit 0 the symbol $W$ is written since $0+1=W$. If it is the digit 1 the symbol $T$ is written since $1+1=T$. No carry is processed now. This will be done at a later stage.

- Note in state $q_1$ the digits of the second number are erased from right to left. If all digits of the second number have vanished the $+$-sign remains as rightmost non-blank. If detected in state $q_1$ the marking phase is finished, the $+$-sign is erased and control moves to state $q_8$. There we change capitals back into digits and take a possible carry into account.

- State $q_8$ represents a situation where there is no carry; state $q_9$ represents a situation where there is a carry. The tape head moves right to left over the symbols $Z$, $W$ and $T$. If in $q_8$ a $Z$ is read a 0 is written. If in $q_8$ a $W$ is read a 1 is written. However, if in $q_8$ a $T$ is read a 0 is written, but since a carry is to be remembered control changes to state $q_9$. Similarly, in state $q_9$ on symbol $Z$ the symbol 1 is written, because of the carry, but since the carry has been handled control returns to state $q_8$. On symbol $W$ a 0 is written but a carry remains. On symbol $T$ a 1 is written and again a carry remains.

- Both in $q_8$ and in $q_9$ the computation is done if a blank is encountered. Then we have scanned over all of the marked digits, i.e. the string of $Z$'s, $W$'s and $T$'s that replaced the first number. However, in $q_8$ nothing is added; in $q_9$ the carry is processed by writing a final 1 in front of the string of digits written so far.

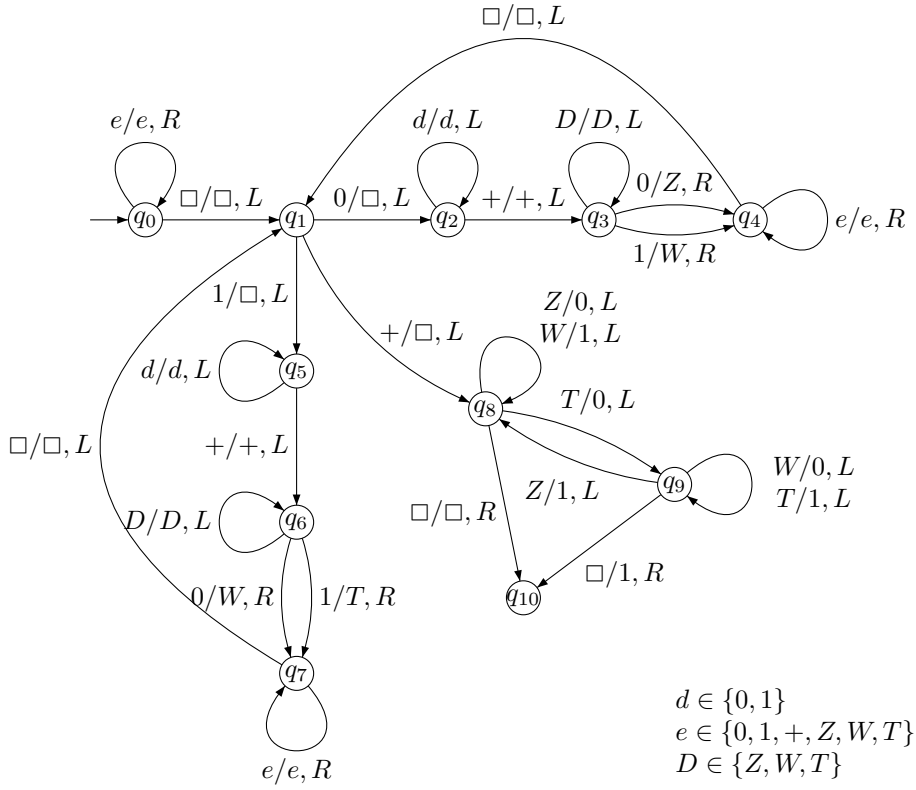



Figure 4.11: A classical Turing machine for binary addition.

# Chapter 5

# Labeled Transitions Systems and Bisimulation

For finite automata and push-down automata we have occassionaly encountered constructs composing these automata in the preceding chapters. In this chapter we introduce constructs to combine so-called labeled transition systems. Exploiting the notion of branching bisimulation we can analyze such composed systems and verify whether they meet the requirements for which they were designed.

## 5.1 Labeled transition systems

In this section we give the formal definition of a labeled transition system, LTS for short. To compare states within an LTS or to compare LTSs, we introduce two notions of equivalence, viz. bisimulation and branching bisimulation and we discuss procedures to decide this.

**Definition 5.1.** A labeled transition system (LTS) is a quadruple $S = (\, Q,\, \Sigma,\, \rightarrow_S,\, q_0\,)$ where

1. $Q$ is a finite or infinite set of states,
2. $\Sigma$ a finite or infinite set of actions, $\tau \notin \Sigma$,
3. $\rightarrow_S \subseteq Q \times \Sigma_\tau \times Q$ a set of transitions or steps, with $\Sigma_\tau = \Sigma \cup \{\tau\}$, and
4. $q_0 \in Q$ is the initial state.

If, with respect to a labeled transition system $S$, $(q, a, q') \in \rightarrow_S$ we write $q \xrightarrow{a}_S q'$. The action $a \in \Sigma$ is referred to as the label of the transition; the state $q$ is called the source state, the state $q'$ is called the target state. The label $\tau$ is also allowed for transitions. In the context of LTSs $\tau$ is called the silent action, although $\tau \notin \Sigma$. In concrete cases an LTS is also called a process.

An LTS resembles an NFA. However, an LTS has no final states. Also an LTS can have an infinite set of states and may have infinitely many different actions. Consequently, we don't compare LTSs by language equivalence, but by a notion called bisimulation instead. However, in the discussion of the decision procedures below we will restrict ourselves to LTSs with finitely many states involving finitely many actions. Such LTSs are called finite LTSs.

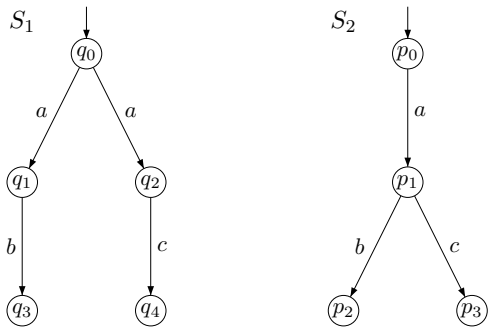



Figure 5.1: Two non-bisimilar LTSs

**Example 5.2.** Consider LTSs $S_1$ and $S_2$ in Figure 5.1. LTS $S_1$ has five states of which $q_0$ is its initial state, $S_1$ has four transitions with labels from the action set $\{a, b, c\}$. LTS $S_2$ has four states of which $p_0$ is its initial state, $S_2$ has three transitions with labels from $\{a, b, c\}$.

If we interpret the LTSs $S_1$ and $S_2$ as NFAs $N_1$ and $N_2$ with final states $q_3, q_4$ and $p_2, p_3$, respectively, we would have $\mathcal{L}(N_1) = \{ab, ac\} = \mathcal{L}(N_2)$. However, the states $q_1$ and $q_2$ of $S_1$ are different from the state $p_1$ of $S_2$: $q_1$ and $q_2$ each only offer one action, $b$ or $c$ exclusively, hence miss the other action, whereas $p_1$ offers both actions.

The point is, in $S_1$ the choice for the action $b$ or the action $c$ is made in state $q_0$ already. In $S_2$ the choice is made at a later moment in execution, viz. in $p_1$ after the action $a$ has taken place.

**Example 5.3.** An LTS for a FIFO queue of capacity 2 that can handle data items of value 0 and 1 is displayed in Figure 5.2. Starting from the empty queue, state $\varepsilon$, either a 0 or a 1 can be input by executing the action $in_0$ or $in_1$ leading to state 0 and state 1, respectively. If, for example, in state 0 a 1 is input by executing the action $in_1$ the next state is state 10 where first a 0 will be output, only the action $out_0$ is possible.

**Example 5.4.** Figure 5.3 shows part of an LTS with an infinite set of states. The LTS represents a counter which can count up a number of steps once, and then can count down the same number of steps (or less). Thus once the LTS goes down, it keeps going

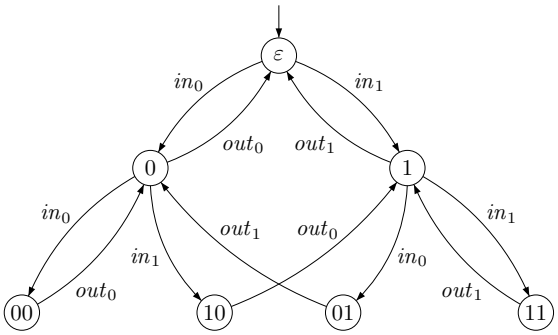



Figure 5.2: LTS of a FIFO queue of capacity 2

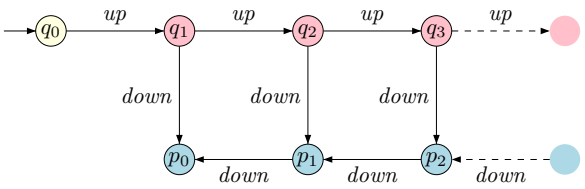



Figure 5.3: Counter: An LTS with infinite set of states

down until it stops. More precisely, we put

$$\begin{aligned} S_{cnt} &= (\, Q_{cnt},\, \{up, down\},\, \rightarrow_{cnt},\, q_0\,) \quad \text{where} \\ Q_{cnt} &= \{\, q_i \mid i \geqslant 0\,\} \cup \{\, p_j \mid j \geqslant 0\,\} \\ \rightarrow_{cnt} &= \{\, (q_i, up, q_{i+1}) \mid i \geqslant 0\,\} \cup \\ &\qquad \{\, (q_k, down, p_{k-1}) \mid k \geqslant 1\,\} \cup \{\, (p_{j+1}, down, p_j) \mid j \geqslant 0\,\} \end{aligned}$$

State $p_0$ is called a deadlock state, since it has no outgoing transitions. (Also in Figure 5.1, states $q_3, q_4$ and states $p_2, p_3$ are deadlock states.)

Next we provide the definition of bisimilarity for two states in an LTS and of bisimilarity between two LTSs. The two notions are also referred to as *strong* bisimilarity to stress that it does not treat $\tau$-actions differently from other actions. We will encounter so-called *branching bisimilarity* later, being a notion of weak bisimilarity.

**Definition 5.5.** (a) Let $S = (\, Q, \Sigma, \rightarrow_S, q_0\,)$ be an LTS. A relation $R \subseteq Q \times Q$ is called a bisimulation relation for $S$ if, for all $q, p \in Q$,

(i) if $R(q,p)$ and $q \xrightarrow{\alpha}_S q'$, for some $\alpha \in \Sigma_\tau$, $q' \in Q$, then $p \xrightarrow{\alpha}_S p'$ for some $p' \in Q$ such that $R(q', p')$;

(ii) if $R(q,p)$ and $p \xrightarrow{\alpha}_S p'$, for some $\alpha \in \Sigma_\tau$, $p' \in Q$, then $q \xrightarrow{\alpha}_S q'$ for some $q' \in Q$ such that $R(q', p')$.

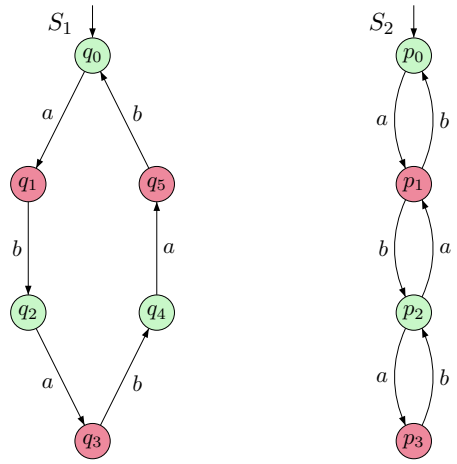



Figure 5.4: Bisimilar states and bisimliar LTSs

Two states $q, p \in Q$ of $S$ are called bisimilar if there exists a bisimulation $R$ for $S$ with $R(q, p)$. Notation $q \leftrightarrows p$.

(b) Let $S_1 = (\, Q_1,\, \Sigma,\, \rightarrow_1,\, q_0\,)$ and $S_2 = (\, Q_2,\, \Sigma,\, \rightarrow_2,\, p_0\,)$ be two LTSs. A relation $R \subseteq Q_1 \times Q_2$ is called a bisimulation relation between $S_1$ and $S_2$ if, for all $q \in Q_1$, $p \in Q_2$,

(i) if $R(q, p)$ and $q \xrightarrow{\alpha}_1 q'$, for some $\alpha \in \Sigma_\tau$, $q' \in Q_1$, then $p \xrightarrow{\alpha}_2 p'$ for some $p' \in Q_2$ such that $R(q', p')$;

(ii) if $R(q, p)$ and $p \xrightarrow{\alpha}_2 p'$, for some $\alpha \in \Sigma_\tau$, $p' \in Q_2$, then $q \xrightarrow{\alpha}_1 q'$ for some $q' \in Q_1$ such that $R(q', p')$.

The LTSs $S_1$ and $S_2$ are called bisimilar if there exists a bisimulation relation $R$ between $S_1$ and $S_2$ with $R(q_0, p_0)$. Notation $S_1 \leftrightarrows S_2$.

The requirements (i) and (ii) in both parts of Definition 5.5 are called *transfer conditions*. Note that the target states $q'$ and $p'$ mentioned in the transfer conditions are required to be related, i.e. $R(q', p')$ for the bisimulation involved.

A state $q$ of an LTS $S_1$ is called bisimilar to a state $p$ of an LTS $S_2$ if there exists a bisimulation relation $R$ for $S_1$ and $S_2$ with $R(q, p)$.

**Example 5.6.** Figure 5.4 shows two LTSs, $S_1$ and $S_2$. Let us focus on LTS $S_1$ first. The relation

$$\begin{aligned} R \;=\; \{\; &(q_0, q_0),\, (q_0, q_2),\, (q_0, q_4),\, (q_2, q_0),\, (q_2, q_2),\, (q_2, q_4),\, (q_4, q_0),\, (q_4, q_2),\, (q_4, q_4),\\ &(q_1, q_1),\, (q_1, q_3),\, (q_1, q_5),\, (q_3, q_1),\, (q_3, q_3),\, (q_3, q_5),\, (q_5, q_1),\, (q_5, q_3),\, (q_5, q_5)\;\} \end{aligned}$$

constitutes a bisimulation relation for this LTS, as can be readily checked. Note that the identity relation $I = \{\, (q_i, q_i) \mid 0 \leqslant i \leqslant 5\,\}$ is included in the bisimulation relation. We claim that the relation $R$ is the largest bisimulation relation for $S_1$.

Comparing the left LTS and the right LTS, these LTSs are bisimilar. A bisimulation for the LTSs is the relation $R'$ given by

$$\begin{aligned} R' \;=\; \{\, &(q_0,p_0),\,(q_0,p_2),\,(q_2,p_0),\,(q_2,p_2),\,(q_4,p_0),\,(q_4,p_2),\\ &(q_1,p_1),\,(q_1,p_3),\,(q_3,p_1),\,(q_3,p_3),\,(q_5,p_1),\,(q_5,p_3)\,\} \end{aligned}$$

To construct a bisimulation relation between $S_1$ and $S_2$ we can argue like this: the initial states $q_0$ and $p_0$ should be related. But then also the states $q_1$ and $p_1$, since these are the only states that can be reached from $q_0$ and $p_0$ by an $a$-action. Next, in view of the action $b$, the state $q_2$ must be related to both $p_0$ and $p_2$; $q_2$ is the only $b$-successor of $q_1$ that can match $p_0$ and $p_2$. Reasoning further like this leads to the conclusion that for a bisimulation relation $R'$ the state $q_0$ is to be related to both $p_0$ and $p_2$. Note, by the requirement $R' \subseteq Q_1 \times Q_2$, no state on the left is related by $R'$ to a state on the left and likewise for states on the right.

We next introduce a notion of a coloring scheme to prepare the description of a procedure to actually construct bisimulation relations.

Let $Q$ be a fixed set of states. An infinite sequence $(c_n)_{n=0}^{\infty}$ of functions $c_n : Q \to \mathbb{N}$ is called a coloring scheme if, for all $n$, and all $q, p \in Q$, it holds that if $c_{n+1}(q) = c_{n+1}(p)$ then $c_n(q) = c_n(p)$.

**Theorem 5.7.** Let $S = (\,Q,\,\Sigma,\,\rightarrow_S,\,q_0\,)$ be a finite LTS. Suppose $(c_n)_{n=0}^{\infty}$ is a coloring scheme such that $c_{n+1}(q) = c_{n+1}(p)$ iff for all $\alpha \in \Sigma_\tau$ it holds that

(i) if $q \xrightarrow{\alpha}_S q'$ for some $q' \in Q$ then $p \xrightarrow{\alpha}_S p'$ for some $p' \in Q$ such that $c_n(p') = c_n(q')$,

(ii) if $p \xrightarrow{\alpha}_S p'$ for some $p' \in Q$ then $q \xrightarrow{\alpha}_S q'$ for some $q' \in Q$ such that $c_n(q') = c_n(p')$.

Define the relation $R \subseteq Q \times Q$ by $R(q,p) \iff c_n(q) = c_n(p)$ for all $n \geqslant 0$. Then $R$ is a bisimulation relation for $S$.

*Proof.* Suppose $R(q,p)$. Let $\alpha \in \Sigma_\tau$ and $q' \in Q$ be an arbitrary action and state such that $q \xrightarrow{\alpha}_S q'$. We need to find $p' \in Q$ such that $p \xrightarrow{\alpha}_S p'$ and $R(q',p')$.

Since $R(q,p)$ we have that $c_{n+1}(q) = c_{n+1}(p)$ for all $n \geqslant 0$. Choose for each $n$ a state $p'_n$ such that $p \xrightarrow{\alpha}_S p'_n$ and $c_n(q') = c_n(p'_n)$. Consider the set of states $\{\,p'_n \mid n \geqslant 0\,\}$. Since $S$ is finite, the set $Q$ is finite. Therefore, there must be a state $p' \in Q$ such that $p' = p'_n$ for infinitely many $n$.

If $m \geqslant 0$ is an arbitrary number, pick $n \geqslant m$ such that $p'_n = p'$. Then $c_n(q') = c_n(p'_n) = c_n(p')$. Since $(c_n)_{i=0}^{\infty}$ is a coloring scheme and $n \geqslant m$ it holds that $c_m(q') = c_m(p')$. Since $m$ was chosen arbitrarily, it follows that $c_m(q') = c_m(p')$ for all $m \geqslant 0$. Thus $R(q',p')$.

By symmetry if follows that if $R(q,p)$, $\alpha \in \Sigma_\tau$ and $p' \in Q$ are such that $p \xrightarrow{\alpha}_S p'$, we can find $q' \in Q$ such that $q \xrightarrow{\alpha}_S q'$ and $R(q',p')$. We conclude that $R$ is a bisimulation relation for $S$, as was to be shown. □

The reverse of Theorem 5.7 also holds: If $R \subseteq Q \times Q$ is a bisimulation relation for an LTS $S$, then the so-called equivalence closure $\tilde{R}$ of $R$, the smallest equivalence relation containing $R$, is also a bisimulation relation. Moreover, the function $c : Q \to \mathbb{N}$ such that $c(q) = c(p)$ iff $\tilde{R}(q, p)$ induces a constant coloring scheme $(c_n)_{n=0}^{\infty}$ with $c_n = c$ for all $n \geqslant 0$ that satisfies the requirements of Theorem 5.7.

**Example 5.8.** Consider again the LTS at the left of Figure 5.4. In view of Theorem 5.7, we want to construct a sequence of colorings from which a bisimulation can be deduced. We start from an initial coloring $c_0$ which assigns to each state the 'color' 0. Next we check the outgoing transitions and group states accordingly. In this case we then conclude that the sequence of colorings stabilizes.

| | $c_0$ | | $c_1$ | | $c_2$ |
|---|---|---|---|---|---|
| $q_0$ | 0 | $a/0$ | 1 | $a/2$ | 1 |
| $q_1$ | 0 | $b/0$ | 2 | $b/1$ | 2 |
| $q_2$ | 0 | $a/0$ | 1 | $a/2$ | 1 |
| $q_3$ | 0 | $b/0$ | 2 | $b/1$ | 2 |
| $q_4$ | 0 | $a/0$ | 1 | $a/2$ | 1 |
| $q_5$ | 0 | $b/0$ | 2 | $b/1$ | 2 |
| (1) | (2) | (3) | (4) | (5) | (6) |

In the leftmost column (1) we list the states and give them initially the color 0, column (2). Next, column (3), we examine the transitions of the states. For example, state $q_0$ gets the mark $a/0$ as $q_0$ has a transition to state $q_1$ which has currently color 0. If the entry in this column is the same for another state, this state gets the same color in column (4). Looking at the transitions-to-colors again, this does not distinguish further states. Apparently two colors, 1 and 2, suffice. The theorem guarantees that the groups of the same color, i.e. $q_0, q_2, q_4$ vs. $q_1, q_3, q_5$, are groups of bisimilar states.

For the LTS at the right-hand side of Figure 5.4 the reasoning is similar. We get the table

| | $c_0$ | | $c_1$ | | $c_2$ |
|---|---|---|---|---|---|
| $p_0$ | 0 | $a/0$ | 1 | $a/2$ | 1 |
| $p_1$ | 0 | $b/0$ | 2 | $b/1$ | 2 |
| $p_2$ | 0 | $a/0$ | 1 | $a/2$ | 1 |
| $p_3$ | 0 | $b/0$ | 2 | $b/1$ | 2 |
| (1) | (2) | (3) | (4) | (5) | (6) |

Note, e.g. for state $p_1$ in column (3), although $p_1$ admits two transitions on action $b$, one to $p_0$ and one to $p_2$, the two transitions collapse since for $c_0$ the states $p_0$ and $p_2$ are assigned the same color.

The examples illustrate that via this procedure all bisimilar states get assigned the same color eventually, a claim we will not prove here.

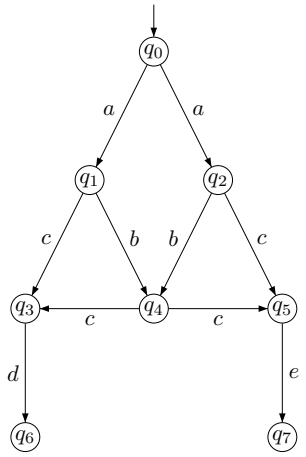



Figure 5.5: Coloring an LTS

**Example 5.9.** Consider the LTS in Figure 5.5. We construct a coloring table to find bisimilar states. In particular, one may wonder if the states $q_1$ and $q_2$ are bisimilar.

We start off with the coloring $c_0 : Q \to \mathbb{N}$, with $Q$ the set of states of the LTS, assigning each state the color 0. Next we determine the transition capabilities with respect to $c_0$, listed in the next column and introduce new colors, a new color for each capability. Note that $c_1$ gives $q_1$ and $q_2$ the same color 2, the ordering of the transitions is irrelevant. Also $q_6$ and $q_7$, both deadlock states, get the same color.

However, coloring $c_2$ distinguishes between color of $q_1$ and $q_2$. This is because $q_1$ has a $c$-transition to state $q_3$ with color 3 but no $c$-transition to state with color 5; for state $q_2$ this is the other way around. For other states, the coloring is updated as well. As a consequence, in the next column, state $q_0$ gets a more refined capability, an $a$-transition to color 8 and an $a$-transition to color 9. The lower part of the coloring is getting stable now. However, due to changes in the coloring it takes to coloring $c_5$ to obtain a stable coloring.

| | $c_0$ | | $c_1$ | | $c_2$ | | $c_3$ | | $c_4$ | | $c_5$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $q_0$ | 0 | $a/0$ | 1 | $a/2$ | 7 | $a/8, a/9$ | 13 | $a/14, a/15$ | 17 | $a/18, a/19$ | 20 | $a/18, a/19$ |
| $q_1$ | 0 | $c/0, b/0$ | 2 | $c/3, b/4$ | 8 | $c/10, b/11$ | 14 | $c/10, b/16$ | 18 | $c/10, b/16$ | 18 | $c/10, b/16$ |
| $q_2$ | 0 | $b/0, c/0$ | 2 | $b/4, c/5$ | 9 | $b/11, c/12$ | 15 | $b/16, c/12$ | 19 | $b/16, c/12$ | 19 | $b/16, c/12$ |
| $q_3$ | 0 | $d/0$ | 3 | $d/6$ | 10 | $d/6$ | 10 | $d/6$ | 10 | $d/6$ | 10 | $d/6$ |
| $q_4$ | 0 | $c/0$ | 4 | $c/3, c/5$ | 11 | $c/10, c/12$ | 16 | $c/10, c/12$ | 16 | $c/10, c/12$ | 16 | $c/10, c/12$ |
| $q_5$ | 0 | $e/0$ | 5 | $e/6$ | 12 | $e/6$ | 12 | $e/6$ | 12 | $e/6$ | 12 | $e/6$ |
| $q_6$ | 0 | – | 6 | – | 6 | – | 6 | – | 6 | – | 6 | – |
| $q_7$ | 0 | – | 6 | – | 6 | – | 6 | – | 6 | – | 6 | – |
| | 1 | | 6 | | 7 | | 7 | | 7 | | 7 | |

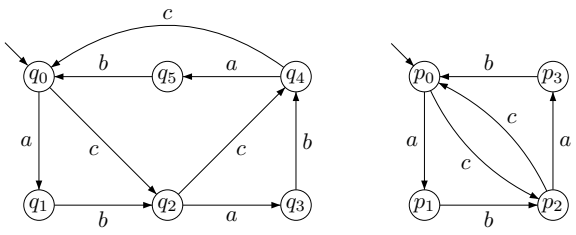



Figure 5.6: Coloring two LTSs

It may happen though, that the procedure doesn't stabilize in colors. However, eventually, because of the finite number of states involved, the colorings are not getting finer any more. At some point the same number of colors will do to give each group of states a distinguishing color. Here we see that only state $q_6$ and $q_7$ are colored identically for coloring $c_2$ and all other states do not have another states uniquely colored. As this remains for coloring $c_3$, we could have stopped there: coloring $c_0$ has 1 color; coloring $c_1$ has 6 colors; colorings $c_2$, $c_3$, $c_4$ and $c_5$ have 7 colors.

A counterpart to Theorem 5.7 for LTSs is available as well.

**Theorem 5.10.** Let $S_1 = (\, Q_1, \Sigma, \to_1, q_0\,)$ and $S_2 = (\, Q_2, \Sigma, \to_2, p_0\,)$ be two finite LTSs. Suppose $(c_n)_{n=0}^{\infty}$ is a coloring scheme of $Q_1 \cup Q_2$ such that, for all $q \in Q_1$ and $p \in Q_2$, $c_{n+1}(q) = c_{n+1}(p)$ iff for all $\alpha \in \Sigma_\tau$ it holds that

- if $q \xrightarrow{\alpha}_1 q'$ for some $q' \in Q_1$ then $p \xrightarrow{\alpha}_2 p'$ for some $p' \in Q_2$ such that $c_n(p') = c_n(q')$,
- if $p \xrightarrow{\alpha}_2 p'$ for some $p' \in Q_2$ then $q \xrightarrow{\alpha}_1 q'$ for some $q' \in Q_1$ such that $c_n(q') = c_n(p')$,

Define the relation $R \subseteq Q_1 \times Q_2$ by $R(q,p) \iff c_n(q) = c_n(p)$ for all $n \geqslant 0$. Then $R$ is a bisimulation relation between $S_1$ and $S_2$.

*Proof.* Left to the reader. □

**Example 5.11.** We construct a combined coloring table for the LTSs of Figure 5.6.

| | | $c_0$ | | $c_1$ | | $c_2$ |
|---|---|---|---|---|---|---|
| $S_2$ | $q_0$ | 0 | $a/0, c/0$ | 1 | $a/2, c/1$ | 3 |
| | $q_1$ | 0 | $b/0$ | 2 | $b/1$ | 4 |
| | $q_2$ | 0 | $a/0, c/0$ | 1 | $a/2, c/1$ | 3 |
| | $q_3$ | 0 | $b/0$ | 2 | $b/1$ | 4 |
| | $q_4$ | 0 | $a/0, c/0$ | 1 | $a/2, c/1$ | 3 |
| | $q_5$ | 0 | $b/0$ | 2 | $b/1$ | 4 |
| $S_1$ | $p_0$ | 0 | $a/0, c/0$ | 1 | $a/2, c/1$ | 3 |
| | $p_1$ | 0 | $b/0$ | 2 | $b/1$ | 4 |
| | $p_2$ | 0 | $a/0, c/0$ | 1 | $a/2, c/1$ | 3 |
| | $p_3$ | 0 | $b/0$ | 2 | $b/1$ | 4 |
| | | 1 | | 2 | | 2 |

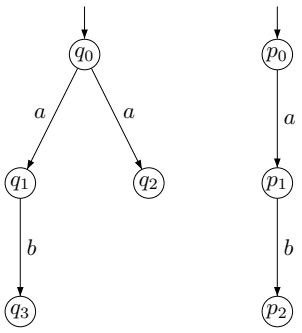



Figure 5.7: Coloring two non-bisimilar LTSs

Note that in going from coloring $c_1$ to $c_2$ new colors have been introduced. However, the number of colors did not increase, it remains 2, hence the grouping of states did not change. There are no states that have the same value for $c_1$ but different values for $c_2$. That is why we can stop after having generated $c_2$.

To compare the LTS on the left to the LTS on the right of Figure 5.6 we compare the initial states $q_0$ and $p_0$. Since they have the same color for the final coloring $c_2$ they are bisimilar. We conclude that the relation $R \subseteq Q_1 \times Q_2$, with $Q_1 = \{\, q_i \mid 0 \leqslant i \leqslant 5 \,\}$ and $Q_2 = \{\, p_j \mid 0 \leqslant j \leqslant 3 \,\}$, given by

$$R(q_i, p_j) \iff i \text{ and } j \text{ both even or both odd}$$

for $0 \leqslant i \leqslant 5$, $0 \leqslant j \leqslant 3$ is a bisimulation between the LTSs and that the two LTSs are bisimilar.

**Example 5.12.** A combined coloring table for the LTSs of Figure 5.7 is the following

| | $c_0$ | | $c_1$ | | $c_2$ | $c_3$ | |
|---|---|---|---|---|---|---|---|
| $q_0$ | 0 | $a/0$ | 1 | $a/2, a/3$ | 4 | $a/5, a/3$ | 7 |
| $q_1$ | 0 | $b/0$ | 2 | $b/3$ | 5 | $b/3$ | 5 |
| $q_2$ | 0 | $-$ | 3 | $-$ | 3 | $-$ | 3 |
| $q_3$ | 0 | $-$ | 3 | $-$ | 3 | $-$ | 3 |
| $p_0$ | 0 | $a/0$ | 1 | $a/2$ | 6 | $a/5$ | 8 |
| $p_1$ | 0 | $b/0$ | 2 | $b/3$ | 5 | $b/3$ | 5 |
| $p_2$ | 0 | $-$ | 3 | $-$ | 3 | $-$ | 3 |
| | 1 | | 3 | | 4 | | 4 |

Since the initial state $q_0$ and $p_0$ of the two LTSs are differently colored we conclude that the two LTSs of Figure 5.7 are not bisimilar.

Consider the two LTSs of Figure 5.8. The two actions $a$ and $b$ of the two LTSs are considered observable, but the silent action $\tau$ of the left LTS isn't. Therefore, from an

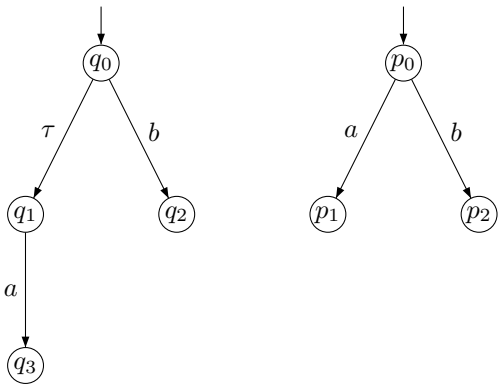



Figure 5.8: Two LTSs that are not branching bisimilar

observer's point of view, it may happen that at some moment in time the left LTS offers an $a$-action only, whereas the right LTS remains offering both an $a$-action and a $b$-action until one of these actions is executed. To capture the internal change in an LTS because of $\tau$-actions we introduce the notion of *branching bisimilarity* of states within an LTS as well as branching bisimilarity of two LTSs. For this we make use, for an LTS $S$ with set of state $Q$, of the notation $q \xrightarrow{\tau}{}^*_S q'$ for states $q, q' \in Q$. We write $q \xrightarrow{\tau}{}^*_S q'$ if for some $n \geqslant 0$, there exist $q_0, q_1, \ldots, q_n \in Q$ such that $q_0 = q$, $q_{i-1} \xrightarrow{\tau}_S q_i$ for $1 \leqslant i \leqslant n$, and $q_n = q'$.

**Definition 5.13.** (a) Let $S = (\, Q,\, \Sigma,\, \rightarrow_S,\, q_0\,)$ be an LTS. A relation $R \subseteq Q \times Q$ is called a branching bisimulation relation for $S$ if, for all $q, p \in Q$,

(i) if $R(q,p)$ and $q \xrightarrow{a}_S q'$, for some $a \in \Sigma$, $q' \in Q$, then $p \xrightarrow{\tau}{}^*_S \bar{p}$ and $\bar{p} \xrightarrow{a}_S p'$ for some $\bar{p}, p' \in Q$ such that $R(q,\bar{p})$ and $R(q',p')$;

(ii) if $R(q,p)$ and $p \xrightarrow{a}_S p'$, for some $a \in \Sigma$, $p' \in Q$, then $q \xrightarrow{\tau}{}^*_S \bar{q}$ and $\bar{q} \xrightarrow{a}_S q'$ for some $\bar{q}, q' \in Q$ such that $R(\bar{q},p)$ and $R(q',p')$;

(iii) if $R(q,p)$ and $q \xrightarrow{\tau}_S q'$ for some $q' \in Q$, then either $p \xrightarrow{\tau}{}^*_S \bar{p}$ and $\bar{p} \xrightarrow{\tau}_S p'$ for some $\bar{p}, p' \in Q$ such that $R(q,\bar{p})$ and $R(q',p')$, or $p \xrightarrow{\tau}{}^*_S p'$ for some $p' \in Q$ such that $R(q,p')$ and $R(q',p')$;

(iv) if $R(q,p)$ and $p \xrightarrow{\tau}_S p'$, for some $p' \in Q$, then either $q \xrightarrow{\tau}{}^*_S \bar{q}$ and $\bar{q} \xrightarrow{\tau}_S q'$ for some $\bar{q}, q' \in Q$ such that $R(\bar{q},p)$ and $R(q',p')$, or $q \xrightarrow{\tau}{}^*_S q'$ for some $q' \in Q$ such that $R(q,p')$ and $R(q',p')$.

Two states $q, p \in Q$ of $S$ are called branching bisimilar if there exists a branching bisimulation relation $R$ for $S$ with $R(q,p)$.

(b) Let $S_1 = (\, Q_1,\, \Sigma,\, \rightarrow_1,\, q_0\,)$ and $S_2 = (\, Q_2,\, \Sigma,\, \rightarrow_2,\, p_0\,)$ be two LTSs. A relation $R \subseteq Q_1 \times Q_2$ is called a branching bisimulation relation between $S_1$ and $S_2$ if, for all $q \in Q_1$, $p \in Q_2$,

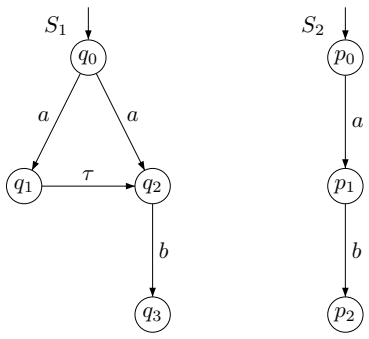



Figure 5.9: Branching bisimilar LTSs

(i) if $R(q,p)$ and $q \xrightarrow{a}_1 q'$, for some $a \in \Sigma$, $q_1' \in Q$, then $p \xrightarrow{\tau}_2^* \bar{p}$ and $\bar{p} \xrightarrow{a}_2 p'$ for some $\bar{p}, p' \in Q_2$ such that $R(q,\bar{p})$ and $R(q',p')$;

(ii) if $R(q,p)$ and $p \xrightarrow{a}_2 p'$, for some $a \in \Sigma$, $p' \in Q_2$, then $q \xrightarrow{\tau}_1^* \bar{q}$ and $\bar{q} \xrightarrow{a}_1 q'$ for some $\bar{q}, q' \in Q_1$ such that $R(p,\bar{q})$ and $R(q',p')$.

(iii) if $R(q,p)$ and $q \xrightarrow{\tau}_1 q'$ for some $q' \in Q_1$, then $p \xrightarrow{\tau}_2^* \bar{p}$ and $\bar{p} \xrightarrow{\tau}_2 p'$ for some $\bar{p}, p' \in Q_2$ such that $R(q,\bar{p})$ and $R(q',p')$, or $p \xrightarrow{\tau}_2^* p'$ for some $p' \in Q_2$ such that $R(q,p')$ and $R(q',p')$;

(iv) if $R(q,p)$ and $p \xrightarrow{\tau}_2 p'$, for some $p' \in Q_2$, then $q \xrightarrow{\tau}_1^* \bar{q}$ and $\bar{q} \xrightarrow{\tau}_1 q'$ for some $\bar{q}, q' \in Q_1$ such that $R(\bar{q},p)$ and $R(q',p')$, or $q \xrightarrow{\tau}_1^* q'$ for some $q' \in Q_1$ such that $R(q',p)$ and $R(q',p')$.

The LTSs $S_1$ and $S_2$ are called bisimilar if there exists a branching bisimulation relation $R$ for $S_1$ and $S_2$ with $R(q_0,p_0)$.

Compared to Definition 5.5 we have a separate transfer condition for both directions dealing with $\tau$-actions. In the transfer conditions for ordinary actions $a \in \Sigma$, the simulating LTS is allowed first to take a number of silent steps before the matching $a$-transition is done. However, there are restrictions: if $p \xrightarrow{\tau}_S^* \bar{p} \xrightarrow{a}_S p'$ is matching $q \xrightarrow{a}_S q'$ then the intermediate state $\bar{p}$ must be related to the source state $q$ and the state $p'$ must be related to the target state $q'$. Symmetrically, requirements are put on the intermediate state $\bar{q}$ and the state $q'$ if $q \xrightarrow{\tau}_S^* \bar{q} \xrightarrow{a}_S q'$ is matching $p \xrightarrow{a}_S q'$.

The $\tau$-transitions are dealt with slightly differently. Now, in addition, a transition $q \xrightarrow{\tau}_S q'$ can be matched by *no* transition of $p$, provided the state $p$ is related to the state $q'$. Note that the reflexive and transitive closure $\xrightarrow{\tau}_S^*$ allows any finite number of steps $\xrightarrow{\tau}_S$, including 0.

**Example 5.14.** Figure 5.9 shows an LTS $S_1$ where the state $q_1$ and the state $q_2$ are branching bisimilar. The state $q_1$ can only do a $\tau$-transition to $q_2$; state $q_2$ can match this by doing nothing. State $q_2$ only has the $b$-transition as possibility; this can be

matched by $q_1$ by first making a $\tau$-move to $q_2$ and then doing the same $b$-transition. It is straightforwardly checked that the relation $R$ given by

$$R = \{\, (q_0, q_0),\ (q_1, q_1), (q_1, q_2),\ (q_2, q_1),\ (q_2, q_2),\ (q_3, q_3) \,\}$$

is a branching bisimulation relation. Note that the identity relation $I = \{\, (q_i, q_i) \mid 0 \leqslant i \leqslant 3 \,\}$ is included in $R$. This can always be done, since a state is always branching bisimilar to itself.

To show that the left and the right LTS of Figure 5.9, LTS $S_1$ and $S_2$, are branching bisimilar LTSs, we consider the relation $R'$,

$$R' = \{\, (q_0, p_0),\ (q_1, p_1), (q_2, p_1),\ (q_3, p_2) \,\}$$

We claim that $R'$ is a branching bisimulation relation. For example to compare $q_1$ and $p_1$: The $\tau$-transition of $q_1$ is matched by no transition of $p_1$. This is OK since we have both $R'(q_1, p_1)$ and $R'(q_2, p_1)$. The $b$-transition of $p_1$ is matched by the $\tau$-transition of $q_1$ to $q_2$ followed by the $b$-transition from $q_2$ to $q_3$. Again this is OK since $R'(q_2, p_1)$ and $R'(q_3, p_2)$.

We have for branching bisimulation a similar result as Theorem 5.7 for strong bisimulation.

**Theorem 5.15.** Let $S = (\, Q,\, \Sigma,\, \rightarrow_S,\, q_0 \,)$ be a finite LTS. Suppose $(c_n)_{n=0}^{\infty}$ is a coloring scheme such that $c_{n+1}(q) = c_{n+1}(p)$ iff for all $a \in \Sigma$ it holds that

- if $q \xrightarrow{a}_S q'$ for some $q' \in Q$ then $p \xrightarrow{\tau}_S^* \bar{p}$ and $\bar{p} \xrightarrow{a}_S p'$ for some $\bar{p}, p' \in Q$ such that $c_n(q) = c_n(\bar{p})$ and $c_n(q') = c_n(p')$,
- if $p \xrightarrow{a}_S p'$ for some $p' \in Q$ then $q \xrightarrow{\tau}_S^* \bar{q}$ and $\bar{q} \xrightarrow{a}_S q'$ for some $\bar{q}, q' \in Q$ such that $c_n(\bar{q}) = c_n(p)$ and $c_n(q') = c_n(p')$, and
- if $q \xrightarrow{\tau}_S q'$ for some $q' \in Q$ then $p \xrightarrow{\tau}_S^* \bar{p}$ and $\bar{p} \xrightarrow{\tau}_S p'$ for some $\bar{p}, p' \in Q$ such that $c_n(q) = c_n(\bar{p})$ and $c_n(q') = c_n(p')$, or $p \xrightarrow{\tau}_S^* p'$ for some $p' \in Q$ such that $c_n(q) = c_n(p')$ and $c_n(q') = c_n(p')$,
- if $p \xrightarrow{\tau}_S p'$ for some $p' \in Q$ then $q \xrightarrow{\tau}_S^* \bar{q}$ and $\bar{q} \xrightarrow{\tau}_S q'$ for some $\bar{q}, q' \in Q$ such that $c_n(\bar{q}) = c_n(p)$ and $c_n(q') = c_n(p')$, or $q \xrightarrow{\tau}_S^* q'$ for some $q' \in Q$ such that $c_n(q') = c_n(p)$ and $c_n(q') = c_n(p')$.

Define the relation $R \subseteq Q \times Q$ by $R(q, p) \iff c_n(q) = c_n(p)$ for all $n \geqslant 0$. Then $R$ is a bisimulation relation for $S$.

*Proof.* Omitted. □

The proof of Theorem 5.15 goes along the same lines as the proof of Theorem 5.7. However, the proof is technically slightly more involved because of the variation of the transfer conditions for branching bisimilarity. E.g., one needs to argue that there are

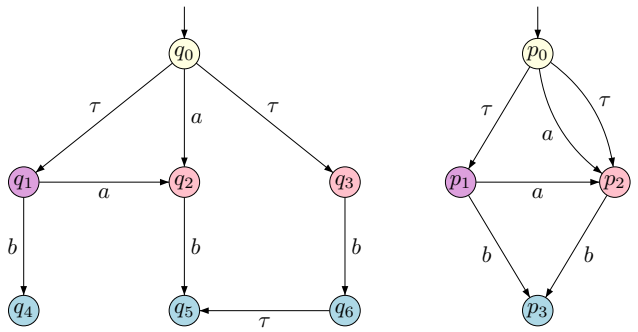



Figure 5.10: Coloring LTSs with internal steps

only finitely many pairs $(\bar{p}, p') \in Q \times Q$ and that one can assume that in infinitely many cases the one or the other form of the transfer conditions for silent steps applies.

As for Theorem 5.7 also the reverse of Theorem 5.15 holds: If two states are branching bisimilar we can construct a coloring scheme that always give the two states the same color. We will neither state nor prove this theorem here.

The parallel of Theorem 5.10 for branching bisimulation is available as well.

**Theorem 5.16.** Let $S_1 = (\, Q_1, \Sigma, \rightarrow_1, q_0 \,)$ and $S_2 = (\, Q_2, \Sigma, \rightarrow_2, p_0 \,)$ be two finite LTSs. Suppose $(c_n)_{n=0}^{\infty}$ is a coloring scheme for $Q_1 \cup Q_2$ such that, for all $q \in Q_1$ and $p \in Q_2$, $c_{n+1}(q) = c_{n+1}(p)$ iff for all $\alpha \in \Sigma_\tau$ it holds that

- if $q \xrightarrow{\alpha}_1 q'$ for some $q' \in Q_1$ then $p \xrightarrow{\tau}_2^* \bar{p}$ and $\bar{p} \xrightarrow{\alpha}_2 p'$ for some $\bar{p}, p' \in Q_2$ such that $c_n(q) = c_n(\bar{p})$ and $c_n(q') = c_n(p')$,
- if $p \xrightarrow{\alpha}_2 p'$ for some $p' \in Q_2$ then $q \xrightarrow{\tau}_1^* \bar{q}$ and $\bar{q} \xrightarrow{\alpha}_1 q'$ for some $\bar{q}, q' \in Q_1$ such that $c_n(\bar{q}) = c_n(p)$ and $c_n(q') = c_n(p')$, and
- if $q \xrightarrow{\tau}_1 q'$ for some $q' \in Q_1$ then $p \xrightarrow{\tau}_2^* \bar{p}$ and $\bar{p} \xrightarrow{\tau}_S p'$ for some $\bar{p}, p' \in Q_2$ such that $c_n(q) = c_n(\bar{p})$ and $c_n(q') = c_n(p')$, or $p \xrightarrow{\tau}_2^* p'$ for some $p' \in Q_2$ such that $c_n(q) = c_n(p')$ and $c_n(q') = c_n(p')$,
- if $p \xrightarrow{\tau}_2 p'$ for some $p' \in Q_2$ then $q \xrightarrow{\tau}_1^* \bar{q}$ and $\bar{q} \xrightarrow{\tau}_1 q'$ for some $\bar{q}, q' \in Q_1$ such that $c_n(\bar{q}) = c_n(p)$ and $c_n(q') = c_n(p')$, or $q \xrightarrow{\tau}_1^* q'$ for some $q' \in Q_1$ such that $c_n(q') = c_n(p)$ and $c_n(q') = c_n(p')$.

Define the relation $R \subseteq Q_1 \times Q_2$ by $R(q,p) \iff c_n(q) = c_n(p)$ for all $n \geqslant 0$. Then $R$ is a bisimulation relation between $S_1$ and $S_2$.

*Proof.* Omitted as well. □

**Example 5.17.** We construct a coloring scheme for the LTSs of Figure 5.10. We need to take care of the $\tau$-transitions. On the one hand, some $\tau$-transitions extend the capabilities of a state; on the other hand, some $\tau$-transitions do not add to the capability.

We will first treat the states $q_0$ to $q_6$ of the left LTS. Initially all states get assigned the color 0. Next we examine the capabilities. State $q_0$ can make a direct transition labeled $a$ to $q_2$, of color 0. But, by first making a $\tau$-transition to $q_1$ it can also make a $b$-transition to $q_4$ of color 0 too. Similarly, $q_0$ can make a combined $b$ transition to $q_6$. The $\tau$-transitions to $q_1$ and $q_3$ are put in parentheses; these do not add to the capability of $q_0$ since it is a $\tau$-transition to the same color as $q_0$ has presently.

Likewise, the $\tau$-transition of $q_6$ to $q_5$ is not counted for. Therefore, $q_6$ is not distinguished by $c_1$ (nor by any coloring later) from the deadlock states $q_4$ and $q_5$. For the rest, the scheme is filled in as before in the case of strong bisimilarity.

Given coloring $c_1$ state $q_0$, of color 1, has a $\tau$-transition to a state of a different color, viz. to $q_3$ of color 2. Therefore, the $\tau$-transition is included in the capabilities of $q_0$. Since the state $q_1$ does not provide a $\tau$-transition to a state of color 2 the next coloring, coloring $c_2$, distinguishing between $q_0$ and $q_1$. Intuitively, while $q_0$ can 'forget' is potential of doing an action $a$ by the silent transition to $q_3$, state $q_1$ cannot do such. State $q_1$ keeps the potential of doing either an $a$ or a $b$ until one of these actions is taken.

| | $c_0$ | | $c_1$ | | $c_2$ | | $c_3$ |
|---|---|---|---|---|---|---|---|
| $q_0$ | 0 | $a/0, b/0, (\tau/0)$ | 1 | $a/2, b/3, \tau/2, (\tau/1)$ | 4 | $a/6, \tau/5, \tau/6$ | 7 |
| $q_1$ | 0 | $a/0, b/0$ | 1 | $a/2, b/3$ | 5 | $a/6, b/3$ | 8 |
| $q_2$ | 0 | $b/0$ | 2 | $b/3$ | 6 | $b/3$ | 6 |
| $q_3$ | 0 | $b/0$ | 2 | $b/3$ | 6 | $b/3$ | 6 |
| $q_4$ | 0 | – | 3 | – | 3 | – | 3 |
| $q_5$ | 0 | – | 3 | – | 3 | – | 3 |
| $q_6$ | 0 | $(\tau/0)$ | 3 | $(\tau/3)$ | 3 | $(\tau/3)$ | 3 |
| $p_0$ | 0 | $a/0, b/0, (\tau/0)$ | 1 | $a/2, b/3, \tau/2, (\tau/1)$ | 4 | $a/6, \tau/5, \tau/6$ | 7 |
| $p_1$ | 0 | $a/0, b/0$ | 1 | $a/2, b/3$ | 5 | $a/6, b/3$ | 8 |
| $p_2$ | 0 | $b/0$ | 2 | $b/3$ | 6 | $b/3$ | 6 |
| $p_3$ | 0 | – | 3 | – | 3 | – | 3 |
| | 1 | | 3 | | 5 | | 5 |

## 5.2 Interaction

In this section we introduce ways to compose and transform LTSs. We introduce the notion of a parallel composition in the context of a communication function, and we introduce the encapsulation and abstraction operators that are used to simplify an LTS while maintaining its essential behaviour.

Consider the three LTSs $S$, $T$ and $S \parallel_\gamma T$ given in Figure 5.11. The left LTS $S$ with initial state $s_0$ represents a webshop. When an order comes in, the order can be

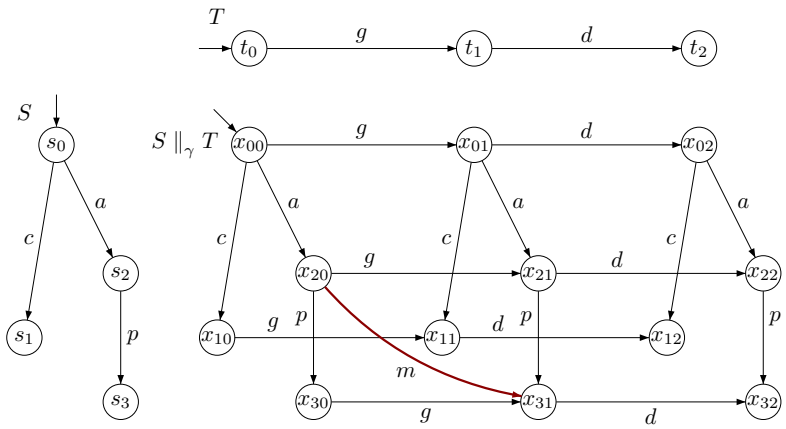



Figure 5.11: An interaction example: webshop $S$, transporter $T$ and system $S \parallel_\gamma T$

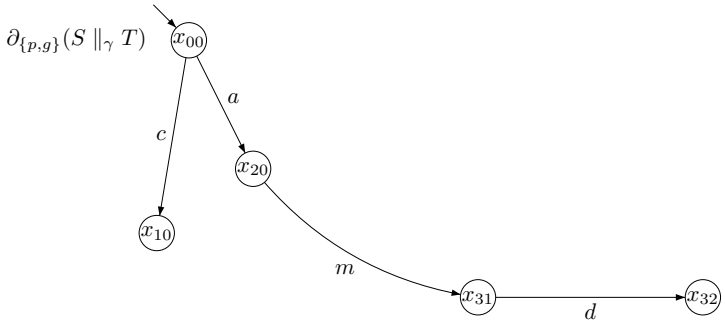



Figure 5.12: Encapsulated LTS: $\partial_{\{p,g\}}(S \parallel_\gamma T)$

accepted, action $a$, and put forward for transport, action $p$, or the order can be canceled, action $c$. The topmost LTS $T$ on the right with initial state $t_0$ represents the transporter. It can get an item, action $g$, and next deliver the item, action $d$.

Of course for a successful interaction of the two LTSs, the put action $p$ and the get action $g$ need to be in sync. To this end we introduce a communication function $\gamma$; we write $\gamma(p, g) = m$ to indicate that the synchronized execution of the action $p$ and the action $g$ yields the action $m$, indicating a move of an item from the shop to the transporter. (Implicitly, we also have $\gamma(g, p) = m$, since communication functions are required to be symmetric in their arguments.) We then put the LTS $S$ and LTS $T$ in parallel taking the communication function $\gamma$ into account, notation $S \parallel_\gamma T$. This is the LTS at the bottom, on the right of Figure 5.11. Note the transition labeled with the action $m$ from state $x_{20}$ to state $x_{31}$.

Still, in the LTS $S \parallel_\gamma T$, putting and getting can occur independently. We would like to enforce that an action $p$ cannot be performed if no action $g$ is present, and vice versa. For this we have an encapsulation operator $\partial_H$, for a set of actions $H$. Here we take

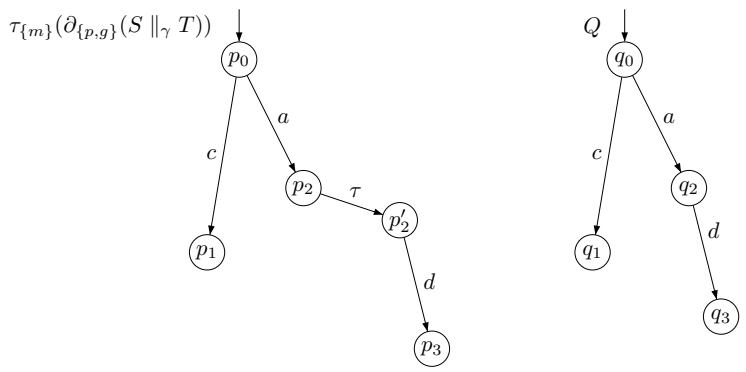



Figure 5.13: Abstracted LTS $\tau_{\{m\}}(\partial_{\{p,g\}}(S \parallel_\gamma T))$ and expected LTS

$H = \{p, g\}$. Figure 5.12 depicts the LTS $\partial_{\{p,g\}}(S \parallel_\gamma T)$ where single actions $p$ and $g$ are not allowed. As a result a number of states have become unreachable from the initial state and are not displayed for that reason. For example, state $x_{01}$ which could only be reached via a $g$-action. As an unpaired $g$-action is not allowed in the LTS $\partial_{\{p,g\}}(S \parallel_\gamma T)$ the state $x_{01}$ will not occur anymore (as many of its successors). As a consequence of the application of encapsulation, the number of (reachable) states has reduced significantly.

We even go one step further. The move, the action labeled $m$, from the shop to the transporter can be considered internal to the LTS. One may argue that for a customer such an action may be irrelevant. We can express this by hiding the action $m$ by means of applying the abstraction operator. The abstraction operator $\tau_I$ for a set of actions $I$ renames the actions occurring in $I$ into the silent action $\tau$. The advantage then is that we can apply branching bisimulation to compare the resulting LTS with another LTS, or to better see the resulting behaviour. In the case of the example we see that the LTS $\tau_{\{m\}}(\partial_{\{p,g\}}(S \parallel_\gamma T))$ is branching bisimilar to the LTS $Q$ with initial state $q_0$, depicted next to it in Figure 5.13. The latter LTS expresses that an order is either canceled, or accepted and delivered next.

We formalize the above with the following definitions.

**Definition 5.18.** Suppose $S_1 = (\, Q_1,\, \Sigma_1,\, \rightarrow_1,\, q_0\,)$ and $S_2 = (\, Q_2,\, \Sigma_2,\, \rightarrow_2,\, p_0\,)$ are two LTSs. Let $\Sigma_0$ and $\Sigma$ be two sets of actions such that $\Sigma_0 \subseteq \Sigma_1 \cup \Sigma_2 \subseteq \Sigma$.

(a) A function $\gamma : \Sigma_0 \times \Sigma_0 \to \Sigma$ is called a communication function for $S_1$ and $S_2$, if, for all $a, b \in \Sigma_0$, it holds that $\gamma(a, b) = \gamma(b, a)$.

(b) If $\gamma : \Sigma_0 \times \Sigma_0 \to \Sigma$ is a communication function for $S_1$ and $S_2$, then the LTS $S_1 \parallel_\gamma S_2$ is given by $S_1 \parallel_\gamma S_2 = (\, Q_1 \times Q_2,\, \Sigma,\, \rightarrow,\, (q_0, p_0)\,)$ where

- $(q, p) \xrightarrow{\alpha} (q', p)$ if $q \xrightarrow{\alpha}_1 q'$ for $\alpha \in \Sigma_1 \cup \{\tau\}$;
- $(q, p) \xrightarrow{\alpha} (q, p')$ if $p \xrightarrow{\alpha}_2 p'$ for $\alpha \in \Sigma_2 \cup \{\tau\}$;

- $(q,p) \xrightarrow{c} (q',p')$ if $q \xrightarrow{a}_1 q'$, $p \xrightarrow{b}_2 p'$, and $\gamma(a,b) = c$ for $a,b \in \Sigma_0$, $c \in \Sigma$;
- $(q,p) \xrightarrow{\tau} (q',p')$ if $q \xrightarrow{\tau}_1 q'$, $p \xrightarrow{\tau}_2 p'$.

If $\gamma$ is a communication function for LTSs $S_1$ and $S_2$, and also for $S_2$ and an LTS $S_3$, then it holds that $(S_1 \parallel_\gamma S_2) \parallel_\gamma S_3$ amounts to the same LTS as the grouping $S_1 \parallel_\gamma (S_2 \parallel_\gamma S_3)$ (up to isomorphy of LTSs, which we do not detail here).

**Definition 5.19.** Let $S = (\, Q,\, \Sigma,\, \rightarrow_S,\, q_0\,)$ be an LTS.

(a) Suppose $H \subseteq \Sigma$ is a subset of actions of $S$. Then the LTS $\partial_H(S)$, the encapsulation of $S$ with respect to $H$, is given by $\partial_H(S) = (\, Q,\, \Sigma,\, \rightarrow',\, q_0\,)$ where $q \xrightarrow{\alpha}' q'$ if $\alpha \notin H$ and $q \xrightarrow{\alpha} q'$ for $\alpha \in \Sigma_\tau$.

(b) Suppose $I \subseteq \Sigma$ is a subset of actions of $S$. Then the LTS $\tau_I(S)$, the abstraction of $S$ with respect to $I$, is given by $\tau_I(S) = (\, Q,\, \Sigma,\, \rightarrow',\, q_0\,)$ where

- $q \xrightarrow{\alpha}' q'$ if $\alpha \notin I$ and $q \xrightarrow{\alpha} q'$ for $\alpha \in \Sigma_\tau$
- $q \xrightarrow{\tau}' q'$ if $q \xrightarrow{a} q'$ for some $a \in I$.

The format $\tau_I(\partial_H(S_1 \parallel_\gamma \cdots \parallel_\gamma S_k))$ is encountered frequently in the analysis of concurrent systems. The format is called standard concurrent form.

## 5.3 A mutual exclusion protocol

We consider a simple mutual exclusion protocol, which concerns the exclusive access by components of a system to a shared resource when needed. A component is in its critical section when it is using the shared resource. We consider a protocol due to Peterson. The protocol should guarantee that at most one component of a system is in its critical section and components get access eventually.

The formulation of the protocol usually uses shared variables in order to achieve coordination of components. We have to translate this to labeled transition system as formalism we have adopted here. In order to do this, we model the shared variables as separate processes.

The protocol uses three shared variables $C, D, T$ and two components $P, Q$. The variables $C, D$ are Boolean variables, $T$ (turn) has value $p$ or $q$. Initially, the value of $C, D$ is *false* and the value of $T$ is $p$. The value of $T$ is the component that last started an attempt to enter its critical section. If $C$ is *false*, this signifies $P$ is not in its critical section, likewise $D$ is *false* means $Q$ is not in its critical section. If $P$ intends to enter its critical section, it must assign the value true to $C$ before it checks the value of $D$, to prevent situations in which the value of both variables is true. Similarly for $Q$. Still, situations can arise in which the value of both $C$ and $D$ is true. In order to prevent deadlock in this case, each component checks whether the other one last started an attempt to enter its critical section, and the one of which this check succeeds actually enters its critical section.

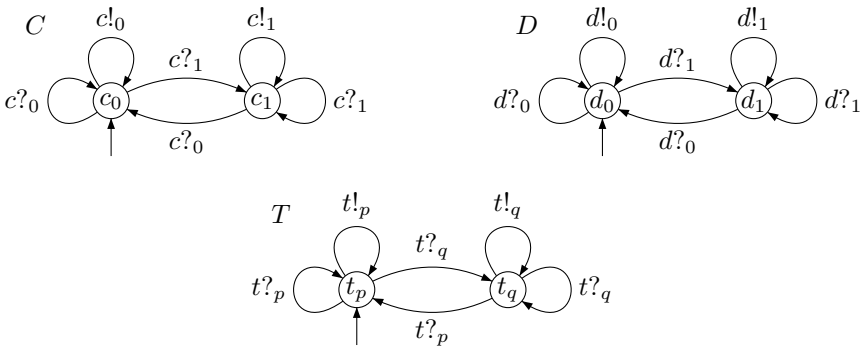



Figure 5.14: LTS for variables $C$, $D$, and $T$

The LTSs for the variables are given in Figure 5.14. In each case, the LTS has two states and provides three actions in each state. For example, with 0 corresponding to the truth value *false*, state $c_0$ represents that the variable $C$ is *false*. In that state the action $c!_0$ can be performed. Likewise, in state $c_1$ corresponding to the variable $C$ having value *true*, the action $c!_1$ is possible. These actions occur when another process inspects the variable. The actions $c?_0$ and $c?_1$ represent new assignments to the variable. The action $c?_0$ is taken if the new value is 0, i.e. *false*; the action $c?_1$ is taken if the new value is 1, i.e. *true*. Note the state change from state $c_0$ to state $c_1$ on action $c?_1$ and, vice versa, from state $c_1$ to state $c_0$ on action $c?_0$. In contrast, when in state $c_0$ on action $c?_0$ the variable remains in state $c_0$; when in state $c_1$ on action $c?_1$ it remains in state $c_1$.

For the components $P$ and $Q$, we draw the LTSs in Figure 5.15. So, focusing on the component $P$, first the variable $C$ is set to *true*, action $c!_1$, indicating that $P$ is approaching the critical section. Then the variable $T$ is set to the value $p$, action $t!_p$, indicating that $P$ was the last one that approached the critical section up to now. Then either when the variable $D$ is *false* or when the variable $T$ is $q$, the non-deterministic choice between the actions $d?_0$ and $t?_q$, the component $P$ enters the critical section, action $enter_p$. Then it leaves, action $leave_p$, and next sets the variable $C$ back to *false* by performing action $c!_0$. The component $Q$ is doing the same, but with the roles of the variables $C$ and $D$ as well as of the components $P$ and $Q$ interchanged.

For the modeling of the setting and testing of the variables $C$, $D$ and $T$, we use the communication function $\gamma$. We choose $\gamma$ such that

$$\gamma(c!_0, c?_0) = c_0 \qquad \gamma(d!_0, d?_0) = d_0 \qquad \gamma(t!_p, t?_p) = t_p$$
$$\gamma(c!_1, c?_1) = c_1 \qquad \gamma(d!_1, d?_1) = d_1 \qquad \gamma(t!_q, t?_q) = t_q$$

Thus $\gamma$ synchronizes two actions $c!_b$ and $c?_b$, one from a component $P$ or $Q$ and one from variable $C$, into one action $c_b$. Here the index $b$ is equal to 0 or 1 referring to the possible boolean values *false* and *true* of the variable $C$. Likewise for the variable $D$. The communication function $\gamma$ also synchronizes two actions $t!_r$ and $t?_r$, one from a component $P$ or $Q$, and one from the variable $T$, into one action $t_r$ where the index $r$ is equal to $p$ or $q$ referring to the component $P$ and $Q$, respectively.

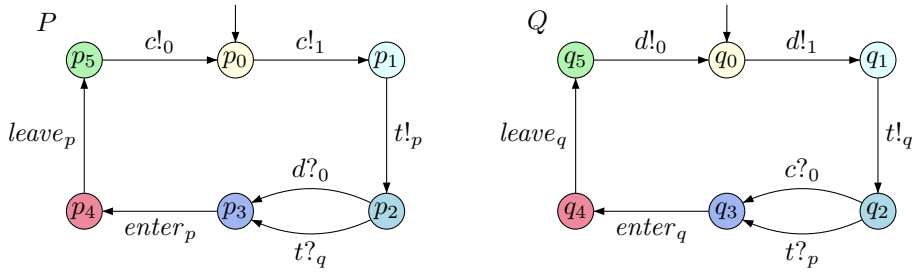



Figure 5.15: Components $P, Q$ of the mutual exclusion protocol

Putting the whole thing together results in the LTS

$$\partial_H(P \parallel_\gamma Q \parallel_\gamma C \parallel_\gamma D \parallel_\gamma T)$$

in Figure 5.16, where $H = \{\, c!_b, c?_b, d!_b, d?_b \mid b = 0, 1 \,\} \cup \{\, t!_r, t?_r \mid r = p, q \,\}$. In each state, we put the values of the variables $C, D, T$ (in this order). If the value of $T$ does not matter, we write an underscore _ as third component. We choose state 00_ as the initial state.

The conclusion is, that the two components cannot be in the critical section at the same time: if one component executes an *enter*-action, then it needs to execute a *leave*-action before the other component can execute an *enter*-action.

Again, all actions from the set $I = \{c_0, c_1, d_0, d_1, t_p, t_q\}$ can be renamed into $\tau$. In this case, branching bisimulation will not remove all $\tau$-steps. The result is shown in Figure 5.17.

We see that an action $enter_p$ is followed by an action $leave_p$ before an action $enter_q$ can occur, possibly interspersed with $\tau$-actions. The symmetric observation can be made for the action $enter_q$ as well. We conclude that the protocol guarantees mutual exclusion indeed. A more refined analysis shows that when component $P$ has reached state $p_2$ it will perform the action $enter_p$ within a finite number of steps. Likewise for component $Q$.

## 5.4 The Alternating Bit Protocol

In this section, we have a look at a communication protocol. This protocol is often referred to as the Alternating Bit Protocol (ABP) in the literature. A communication protocol concerns the transmission of data through an unreliable medium in such a way that – despite the unreliability – no information will get lost. The communication network used in the example is shown in Figure 5.18.

The following describes the components of this network. The component $S$ is the sender, sending data elements $d \in \mathcal{D}$ to the receiver $R$ via the unreliable channel $K$. After having received a certain data element, $R$ will send an acknowledgment to $S$ via channel $L$ which is unreliable as well (in practice, $K$ and $L$ are usually physically

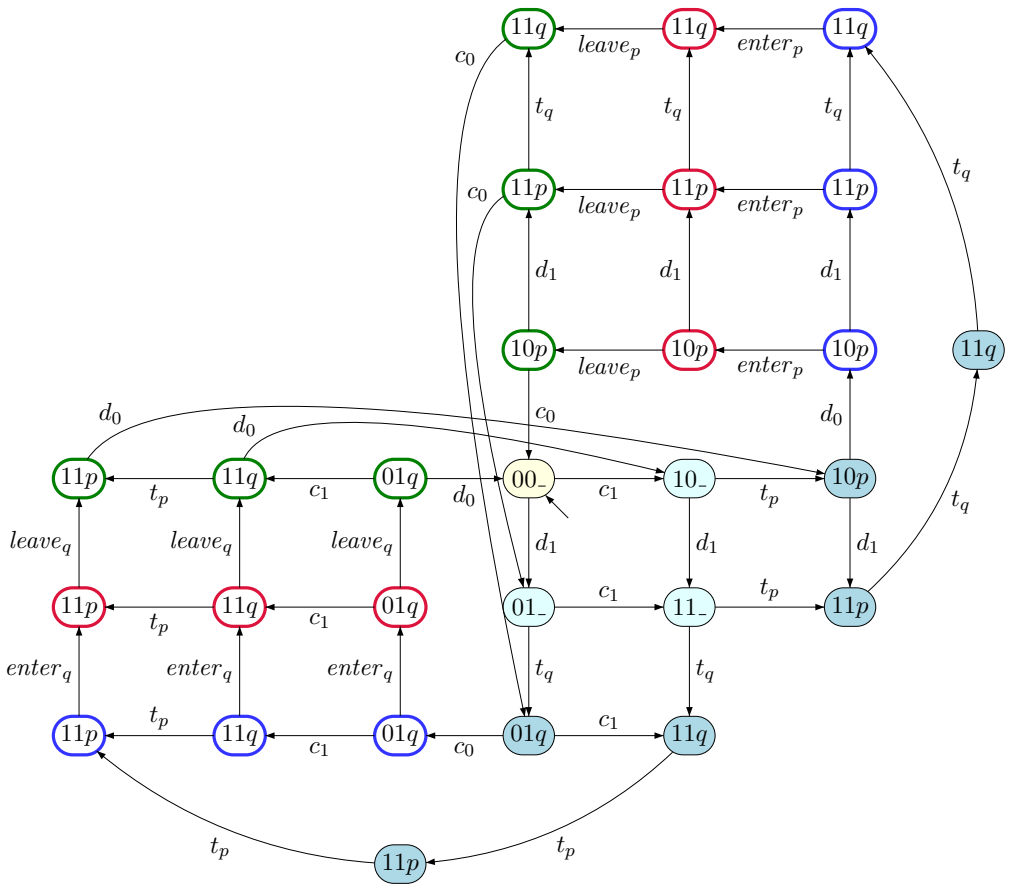



Figure 5.16: Mutual exclusion protocol $\partial_H(P \parallel_\gamma Q \parallel_\gamma C \parallel_\gamma D \parallel_\gamma T)$

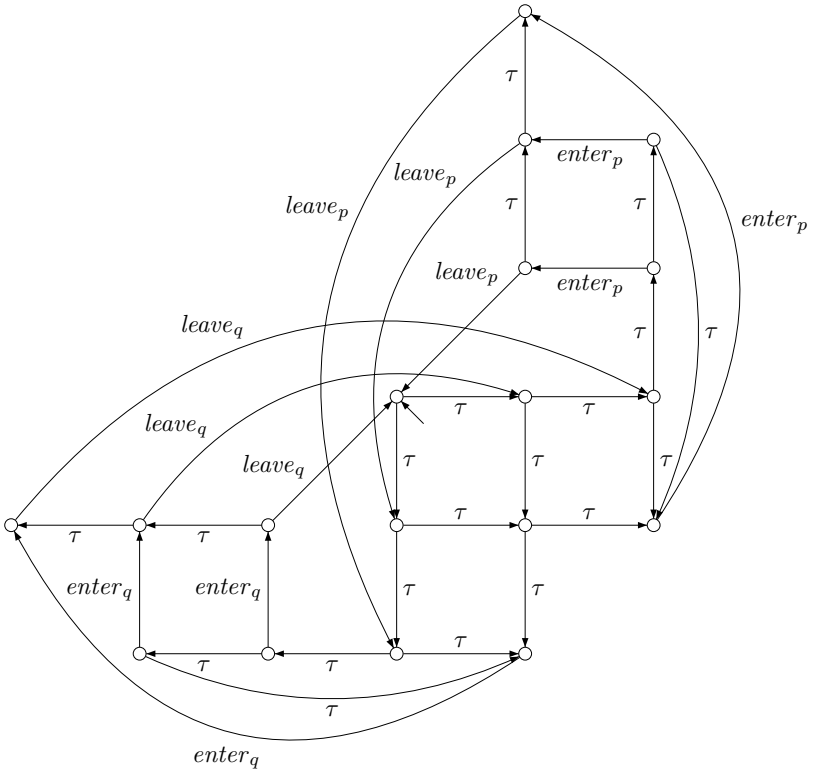



Figure 5.17: Abstracted process $\tau_I(\partial_H(P \parallel_\gamma Q \parallel_\gamma C \parallel_\gamma D \parallel_\gamma T))$

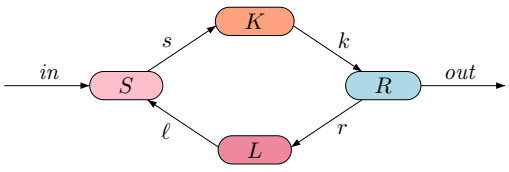
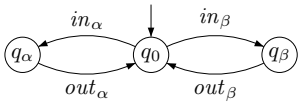



Figure 5.18: Architecture for the ABP

the same medium). The problem now is to define processes $S$ and $R$ such that no information will get lost; that is, the behavior of the entire process, apart from the internal communications, acts as a buffer of capacity 1.

For simplicity we assume the data set $\mathcal{D}$ to consist of two elements, denoted $\alpha$ and $\beta$. The sender $S$ is fed with data items carrying $\alpha$ and $\beta$ expressed by the actions $in_\alpha$ and $in_\beta$. The receiver $R$, when it receives a new data item (in a frame, as we will see) it delivers it via an action $out_\alpha$ or $out_\beta$, dependent on the payload. So, the desired observable behaviour of the entire system is described by the process




A solution can be formulated as follows.

- The sender $S$ reads a data item $d$ at its input and passes on a sequence of frames $\langle d, 0\rangle, \langle d, 0\rangle, \langle d, 0\rangle, \ldots$, i.e. of copies of the data item $d$ tagged with a bit 0 to $K$ until an acknowledgment 0 is received from $L$. Then, the next data item is read, and sent on together with the *alternated* bit, i.e. bit 1; the acknowledgment then is the reception of a 1. The following data element has, in turn, 0 as an appended bit. Thus, 0 and 1 form the alternating bit.

- The process $K$ denotes the data transmission channel, passing on frames of the form $\langle d, 0\rangle, \langle d, 1\rangle$. $K$ may corrupt data, however, passing on $*$ (an error message; thus, it is assumed that the incorrect transmission of $d$ can be recognized, for instance, using a checksum).

- The receiver $R$ gets frames $\langle d, 0\rangle, \langle d, 1\rangle$ and error messages $*$ from $K$, delivering $d$ to its output (if this was not already done earlier), and in response the acknowledgment 0 respectively 1 is sent to $L$.

- The process $L$ is the acknowledgment transmission channel, and passes bits 0 or 1, received from $R$, on to $S$. $L$ is also unreliable, and may send on $*$ instead of a bit 0 or 1.

The processes $S$, $K$, $R$, and $L$ can be specified by means of LTSs. Let $\mathcal{D}$ be the data set $\{\alpha, \beta\}$, define the set of frames by $\mathcal{F} = \{\ \langle\alpha, 0\rangle, \langle\beta, 0\rangle, \langle\alpha, 1\rangle, \langle\beta, 1\rangle\ \}$, and let $\tau$ be the silent action. The LTSs of the channels $K$ and $L$ are given in Figure 5.19. In these LTSs, the action $\tau$ serves to make the choice non-deterministic: the decision whether or not the frame will be corrupted is internal to the channel, and cannot be influenced by the environment. In branching bisimulation, the treatment of $\tau$ will entail that these description is *not* bisimilar to the ones where the $\tau$-steps are just removed.

The sender $S$ and the receiver $R$ are given in Figure 5.20 and Figure 5.21. From the starting state $s_0$ the sender takes an input $\alpha$ or $\beta$. It sends out the data item in a frame with bit 0. If no correct acknowledgment is received, i.e. bit 1 or error message $*$

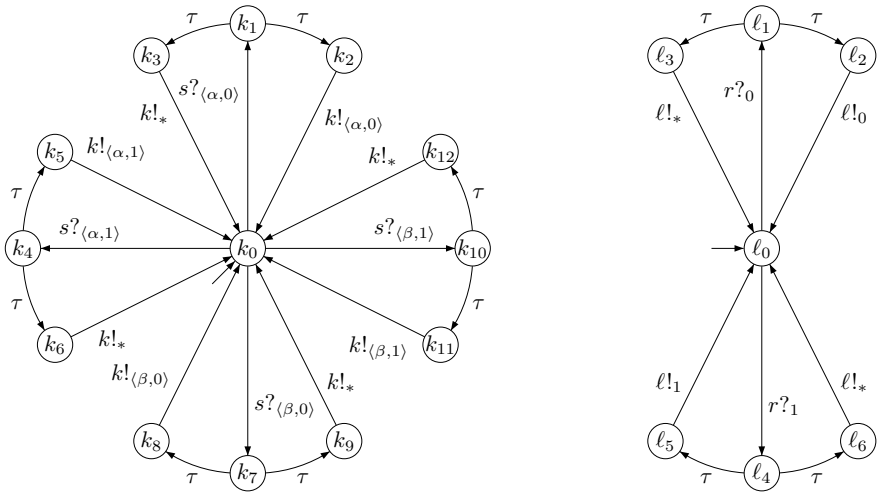



Figure 5.19: LTSs of the channels $K$ and $L$

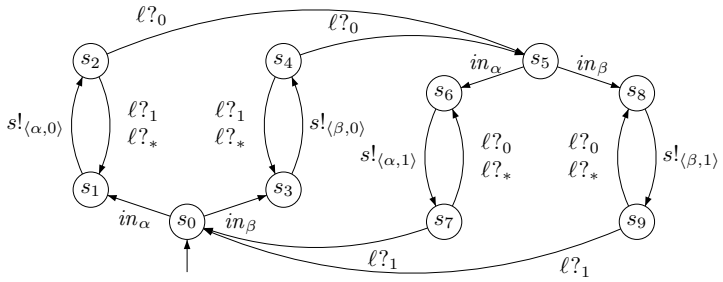



Figure 5.20: LTS of sender $S$

is received instead, the frame is resent. If the acknowledgment is received, i.e. bit 0, the sender changes mode and moves to state $s_5$. There it behaves similarly. However, frames are now sent out with bit 1 and acknowledgments are checked for bit 1.

As depicted in Figure 5.21, the receiver reacts on incoming frames. Starting from state $r_0$ it awaits an incoming frame. If the incoming frame is tagged with bit 1 or is an error message, the acknowledgment 1 is repeated. But, if a frame with bit 0 is received, the corresponding payload is output, actions $out_\alpha$ and $out_\beta$, and the receiver changes mode by moving to state $r_3$. There the receiver behaves similarly. However, now bit 0 is repeated as acknowledgment until a frame is received that is tagged with bit 1.

The communication is governed by the communication function $\gamma$. Here $\gamma$ synchronizes actions of the form $s?_x$ with actions of the form $s!_x$, for frames $x$ of the form $\langle\alpha,0\rangle$, $\langle\beta,0\rangle$, $\langle\alpha,1\rangle$, and $\langle\beta,1\rangle$, and likewise for $k!_x$ and $k?_x$. The communication function $\gamma$ also synchronizes actions of the form $r!_b$ and $r?_b$, and actions of the form $\ell!_b$ and $\ell?_b$,

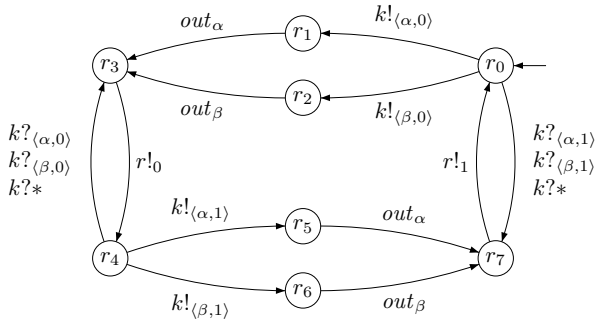



Figure 5.21: LTS of receiver $R$

for $b \in \{0,1\}$. We put

$$\begin{array}{rlr} \gamma(s!_x, s?_x) = s_x, & \gamma(k!_x, k?_x) = k_x & \text{for } x = \langle\alpha,0\rangle, \langle\beta,0\rangle, \langle\alpha,1\rangle, \langle\beta,1\rangle \\ \gamma(r!_b, r?_b) = r_b, & \gamma(\ell!_b, \ell?_b) = \ell_b & \text{for } b = 0,1 \end{array}$$

To enforce synchronization the parallel composition of the components needs to be enclosed by an encapsulation operator that forbids single actions $s?_x$, $s!_x$, $k?_x$, $k!_x$ and $r!_b$ and $r?_b$, $\ell!_b$ and $\ell?_b$, for all frames $x$ and bits $b$. To this end we define

$$\begin{array}{rl} H = & \{\, s!_x,\, s?_x,\, k!_x,\, k?_x \mid x = \langle\alpha,0\rangle, \langle\beta,0\rangle, \langle\alpha,1\rangle, \langle\beta,1\rangle \,\} \cup \\ & \{\, r!_b,\, r?_b,\, \ell!_b,\, \ell?_b \mid b = 0,1 \,\} \end{array}$$

Now an expression for the whole system is

$$\partial_H(S \parallel_\gamma K \parallel_\gamma L \parallel_\gamma R)$$

where $\partial_H$ will block all unmatched !-actions and ?-actions.

Using the rules for the composition of LTSs we can derive the LTS in Figure 5.22 for the complete system for the ABP. In the first half of the LTS, we use a data-element $d$, in the second half $e$. In fact, the LTS should show a branching over $\mathcal{D}$, i.e. both for $\alpha$ and $\beta$ at states $q_0$ and $q_{18}$. This means that the upper left-hand corner and the lower right-hand corner should have outgoing edges $in_\alpha$ and $in_\beta$ that we have depicted here, for a compact presentation, as $in_d$ for the upper left-hand corner and $in_e$ for the lower right-hand corner.

Next, we want to hide the communication over the internal channels, i.e. we want to turn communication involving $s$, $k$, $\ell$, and $r$ into $\tau$. Indeed, all $\tau$-steps can be removed, and we can show branching bisimilarity with the buffer of capacity one. Hence, the analysis shows that the ABP does not loose information, as required.

Notice that the notion of branching bisimilarity embodies a form of *fairness*: after a number of failing communications, eventually a successful communication will take place, the channel cannot be totally defective.

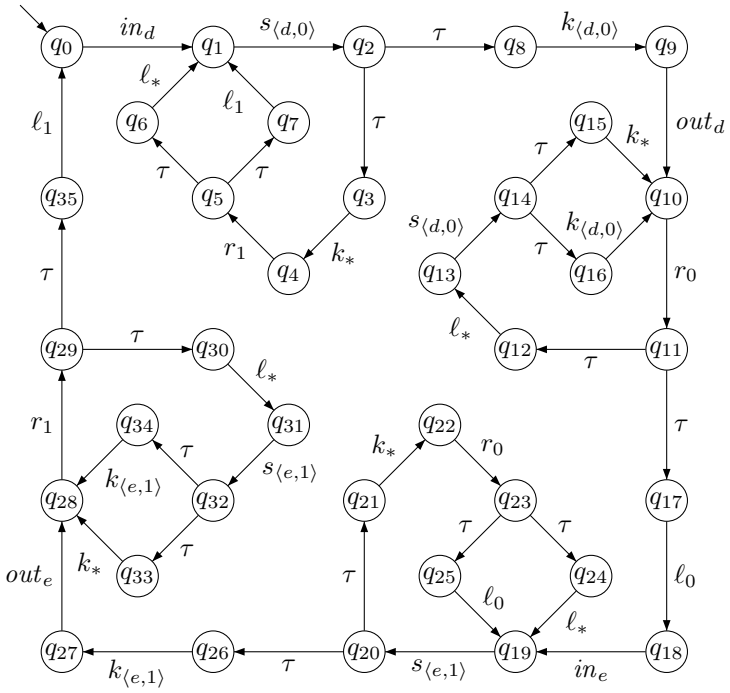



Figure 5.22: LTS of the ABP $\partial_H(S \parallel_\gamma K \parallel_\gamma L \parallel_\gamma R)$